

# Dokumentation

Daniel Friedrich

Verbundprojekt 2018/2019

Daniel Friedrich

## Verbundprojekt 2018/2019

Abschlusspräsentation eingereicht im Rahmen des Verbundprojektes

im Studiengang Master of Science Automatisierung  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Ulfert Meiners  
Zweitgutachter: Prof. Dr.-Ing. Jochen Maaß, Prof. Dr.-Ing. Michael Röther

Eingereicht am: 25. Mai 2019

**Daniel Friedrich**

**Thema der Arbeit**

Verbundprojekt 2018/2019

**Stichworte**

Smart Kamera, Objekterkennung, Farbdetektion, Gehäusekonstruktion, Pixy-Cam, Raspberry Pi

**Kurzzusammenfassung**

Im Rahmen dieser Arbeit werden zwei Kamerasysteme verglichen, welche farbige Objekte auf einem Laufband erkennen und deren Farbe auswerten. Über verschiedene Schnittstellen wird die Auswertung ausgegeben. Dazu werden Anwendungen entworfen, welche die Auswertung auf einem der Systeme ermöglichen. Die Arbeit beschreibt außerdem die konkrete Implementierung des Quellcodes der entworfenen Anwendungen für eines der Kamerasysteme. Ein weiterer zentraler Punkt der Arbeit ist der Entwurf einer geeigneten Halterung für die Kamerasysteme.

**Daniel Friedrich**

**Title of Thesis**

Verbundprojekt 2018/2019

**Keywords**

Smart Camera, Objecttracking, Colordetection, Construction, PixyCam, Raspberry Pi

**Abstract**

In this thesis, two camera systems are compared, which recognize colored objects on a conveyor belt and evaluate their color. The evaluation is output via various interfaces. For this purpose, applications are designed, which enable the evaluation on one of the systems. The thesis also describes the implementation of the source code of the designed applications for one of the camera systems. Another point of the work is the design of a suitable holder for the camera systems.

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Vorhaben . . . . .	1
<b>2. Grundlagen und Konzepte</b>	<b>3</b>
2.1. Qt . . . . .	3
2.1.1. Eventbasiertes System . . . . .	3
2.1.2. Sockets . . . . .	3
2.1.3. Datenbank in Qt . . . . .	3
2.1.4. State Machines in Qt . . . . .	3
<b>3. Aufgabenstellung</b>	<b>4</b>
<b>4. Schnittstellen</b>	<b>5</b>
4.1. Kommunikation zu Robotinos . . . . .	5
4.1.1. Sequenzdiagramm . . . . .	5
4.1.2. Telegramme . . . . .	6
4.1.3. Fehlertypen und Behebung . . . . .	9
<b>5. Implementierung</b>	<b>10</b>
5.1. Programmstruktur . . . . .	10
5.2. Database-Handler . . . . .	10
5.2.1. Konstruktor, Initialisierung und Sendefunktion . . . . .	10
5.2.2. Getter . . . . .	12
5.2.3. Updates . . . . .	12
5.2.4. Setter . . . . .	12
5.3. UDP-Handler . . . . .	13
5.3.1. UDP-Handler für Navigationsrechner . . . . .	13
5.3.2. UDP-Handler für Robotino . . . . .	14
5.4. Robotino . . . . .	16
5.4.1. Werte aktualisieren . . . . .	17
5.4.2. Visualisierungsfunktionen . . . . .	17
5.5. Auftrag, Prozess, Prozessschritt . . . . .	18
5.5.1. Prozessschritt . . . . .	18
5.5.2. Prozess . . . . .	18
5.5.3. Auftrag . . . . .	20
5.6. Fertigungsplanung . . . . .	20
5.6.1. Initialisierung . . . . .	21
5.6.2. Roboter Statusänderungen . . . . .	24
5.6.3. Hard-Code Funktionen . . . . .	25
5.6.4. Zustandsdiagramm . . . . .	25
5.6.5. State Machine Implementierung . . . . .	26

5.6.6. Shuffle-Algorithmus . . . . .	29
5.7. Mainwindow . . . . .	30
5.8. Main . . . . .	30
5.9. Weitere Klassen . . . . .	30
<b>6. Visualisierung</b>	<b>32</b>
6.1. Design . . . . .	32
6.2. Struktur . . . . .	33
6.3. Live-View . . . . .	33
6.3.1. Robotino . . . . .	34
6.3.2. Parkplätze und Ladestationen . . . . .	36
6.3.3. Stationen . . . . .	36
6.4. Auftragsübersicht . . . . .	37
6.5. Tab-View . . . . .	37
6.5.1. Log-View . . . . .	37
6.5.2. Manual Control . . . . .	37
6.5.3. Timestamp-Area . . . . .	37
6.5.4. Hard-Code Bereich . . . . .	38
6.6. Roboterstatus . . . . .	38
6.7. Prozesseingabe . . . . .	38
6.8. Auftragseingabe . . . . .	39
6.9. Benutzerinteraktion . . . . .	39
6.9.1. Tooltips . . . . .	39
6.10. MainWindow . . . . .	40
<b>7. Simulation</b>	<b>42</b>
7.1. Zustandsdiagramm simulierter Roboter . . . . .	42
<b>8. Datenbank</b>	<b>45</b>
<b>A. Inhalt der CD</b>	<b>48</b>

# 1. Einleitung

Um an einem realen System Abläufe und Prozesse zu simulieren, wird den Studierenden eine Anlage zur Verfügung gestellt, welche hauptsächlich aus Teilen der Firma „Festo“ besteht (folgend: Festo-Transfersystem). An diesem Festo-Transfersystem können die Studierende Laufbänder, Ampelanlage, Lichtschranken, Weichen etc. mithilfe selbst geschriebener Programme ansteuern und auslesen.

Das Laufband kann aufgelegte Objekte transportieren. Dabei kann die aktuelle Position des Objekts durch diverse Lichtschranken erfasst werden. Am Anfang des Laufbands kann mithilfe eines Entfernungssensors die Höhe des Objekts erfasst werden. Mit den vorhandenen Komponenten kann das Festo-Transfersystem allerdings ausschließlich die Position und die Höhe der Objekte bestimmen und nicht deren Farbe, Form, Ausrichtung oder sonstige optische Merkmale.



Abbildung 1.1.: CMUcam5 Pixy

Eine Erweiterung um ein Kamerasystem kann diese Mängel beheben und sogar noch weitere Funktionen und Features hinzufügen.

## 1.1. Vorhaben

blabla

Ähnlich einer Smart Kamera [Sch09], bei der nicht nur das Kamerabild, sondern auch weitere schon verarbeitete Informationen ausgegeben werden sollen, wird

## 1. Einleitung

ein Prozessor benötigt, welcher in der Lage ist Bildverarbeitung durchzuführen. Es wurde als Möglichkeit für ein Prozessorsystem empfohlen, sich mit dem BeagleBone Black und Raspberry PI auseinanderzusetzen. Mithilfe einer Marktrecherche werden zunächst weitere geeignete Kamerasysteme gesucht und anschließend anhand der Eignung sortiert. Dabei wird auch eine geeignete Schnittstelle zu dem Kamerasystem ausgewählt, mit dem die Bilder und Informationen an den PC gesendet werden. Nachdem die beiden geeignetsten Kamerasysteme gewählt und bestellt sind, werden zugehörige Gehäuse und Befestigungen konstruiert und gefertigt. Nebenbei werden erste einfache Testprogramme geschrieben, um die Funktion von den gewählten Kamerasystemen zu gewährleisten. Der erste Prototyp wird dabei voraussichtlich mit Hilfe rapid-prototyping ein Erzeugnis aus dem 3D-Druck sein, um Kamera an der richtigen Position zu halten. Nachdem das Endprodukt montiert ist, folgt eine Auswertung der Kamerabilder. Folgend folgen optional das Erstellen einer Bibliotheksdatei und eine Automation der Kalibrierung und des Weißabgleichs. Der Zugriff auf die Kamera soll gewährleistet sein.



Abbildung 1.2.: PixyCam Gehäusefehler

## 2. Grundlagen und Konzepte

Grundlagen

### 2.1. Qt

#### 2.1.1. Eventbasiertes System

#### 2.1.2. Sockets

<https://doc.qt.io/qt-5/qudpsocket.html> [Qt 19b]

#### 2.1.3. Datenbank in Qt

MySQLLib.dll

<https://doc.qt.io/qt-5/qsqldatabase.html> [Qt 19a]

#### 2.1.4. State Machines in Qt

<https://doc.qt.io/qt-5/statemachine-api.html>

erwähnen, dass nur `entered()` benutzt wird

[Qt 19c]



# 3. Aufgabenstellung

Um an

## 4. Schnittstellen

### Schnittstellen

### 4.1. Kommunikation zu Robotinos

Die Robotinos werden von Gewerk 2 verwaltet und programmiert. Um eine erfolgreiche Kommunikation zwischen Fertigungsplanungsrechner und Robotino zu gewährleisten existieren folgende Vereinbarungen mit Gewerk 2.

Die Kommunikation geschieht über UDP. Da UDP ein verbindungsloses Protokoll ist muss nicht sichergestellt werden dass Robotino oder Fertigungsplanungsrechner immer verbunden sind. Die Nachrichten werden an spezifizierten Ports gesendet und empfangen.

Die in Tabelle 4.1 dargestellte Vereinbarung wurde für die Ports getroffen.

Roboter IP Adresse	Sendeport	Empfangsprot
192.168.0.11	25010	25011
192.168.0.12	25020	25012
192.168.0.13	25030	25013
192.168.0.14	25040	25014
192.168.0.25	25050	25015

Tabelle 4.1.: Portvereinbarung

#### 4.1.1. Sequenzdiagramm

Zunächst wurde eine Struktur entwickelt, in der beschrieben wird, wie die Daten zwischen Robotino und Fertigungsplanungsrechner ausgetauscht werden sollen. Dies wurde anschließend in einem Sequenzdiagramm, Abbildung 4.1, dargestellt.

Dem Sequenzdiagramm ist zu entnehmen, dass der Robotino nach Programmstart zyklisch Daten sendet. Die Zykluszeit beträgt 100 ms. Da die Daten über UDP an einen spezifizierten Port versendet werden ist ein Empfänger nicht zwangsweise erforderlich und die Programme können unabhängig voneinander laufen. Der Inhalt der Daten ist in Abschnitt 4.1.2 beschrieben. Solange der Robotino läuft werden die Daten gesendet. Dadurch kann auch ein Ausfall der Kommunikation festgestellt werden.

#### 4. Schnittstellen

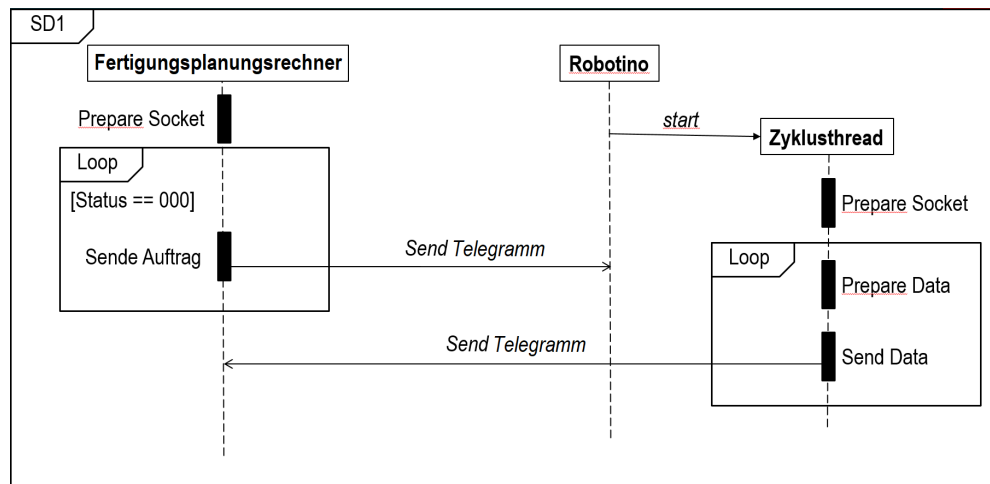


Abbildung 4.1.: Sequenzdiagramm

Auf Seiten des Fertigungsplanungsrechners wird nur bei Bedarf ein Sendevorgang eingeleitet. Erst wenn ein Auftrag an den Robotino gesendet werden soll wird eine Schleife ausgeführt. In der Schleife wird der Auftrag zyklisch an den Robotino gesendet. Die Zykluszeit beträgt hierbei 700 ms. In Kapitel 5.6.5 ist die in einer State-Machine implementierte Schleife beschrieben. Die Schleife kann durch Empfangen einer Statusänderung des Robotinos verlassen werden. Dadurch wird eine erfolgreiche Übertragung des Auftrags sichergestellt werden. Die im Auftrag befindlichen Daten sind in Abschnitt 4.1.2 näher erläutert.

##### 4.1.2. Telegramme

Die Telegramme zwischen Robotino und Fertigungsplanungsrechner bestehen aus einer festgelegten Struktur. Dabei ist sowohl die Länge der Telegramme als auch der Inhalt festgeschrieben. In beiden Telegrammen werden Double-Werte in einem Byte versendet. Jedes Byte muss daher kodiert und dekodiert werden.

##### Telegramm vom Fertigungsplanungsrechner zu Gewerk 2

Der Auftrag, der an den Robotino gesendet wird, enthält 5 Bytes. Eine Aufschlüsselung ist in Tabelle 4.2 dargestellt.

Byte	Inhalt	Beschreibung
1	Auftragsart	1: Transport; 2: Parken; 3: Laden
2	Position 1	z.B. 11
3	Position 2	z.B. 32
4	AliveStatus	1 senden nach Alive Anfrage
5	*Reserved	

Tabelle 4.2.: Telegramm zu Gewerk 2

Das erste Byte klassifiziert die Auftragsart. Diese beschreibt, welche Tätigkeit der Robotino als nächstes tun soll. Auftragsart eins bedeutet, dass der Robotino

#### 4. Schnittstellen

einen Transportauftrag erhalten hat, also ein Werkstück von einer Position zu einer anderen Computer fahren soll. Die Auftragsart 2 zeigt an, dass der Robotino auf einen Parkplatz geschickt wird. Eine Ladefahrt wird mit Auftragsart 3 gekennzeichnet.

Im zweiten und dritten Byte sind die Positionen angegeben, an welche sich der Robotino bewegen soll. Dabei wird Position 2 nur bei Auftragsart 1 befüllt bzw. ausgewertet, da ein Parken und Laden nur eine Zielposition benötigt. Bei Auftragsart 1 jedoch zeigt Position 1 den Arbeitsplatz an, wo das Werkstück abgeholt werden soll und Position 2 den Ablageort des Werkstücks.

Über das vierte Byte kann dem Robotino ein Status-Flag gesendet werden. Sobald der Robotino eine Anfrage zu dem Flag sendet wird eine 1 zurückgesendet. Ansonsten ist der Wert 0. Damit kann der Robotino die Funktionalität der Kommunikation validieren.

Mit dem fünften Byte wird eine einfache Erweiterung des Telegramms ermöglicht, sofern mehr Daten übertragen werden sollen. Am Anfang der Ausarbeitung des Projektes war das vierte Byte ebenfalls ein reserviertes Byte.

#### Positionskodierung

Aus den Positionen, die dem Robotino gesendet werden kann eindeutig bestimmt werden, an welchen Ort der Robotino fahren soll.

Eine Aufschlüsselung der Positionen ist in Abbildung 4.2 dargestellt. Hier ist die Anordnung von Stationen, Parkplätzen und Ladestationen im Raum dargestellt, mitsamt ihrer Positionskodierung.

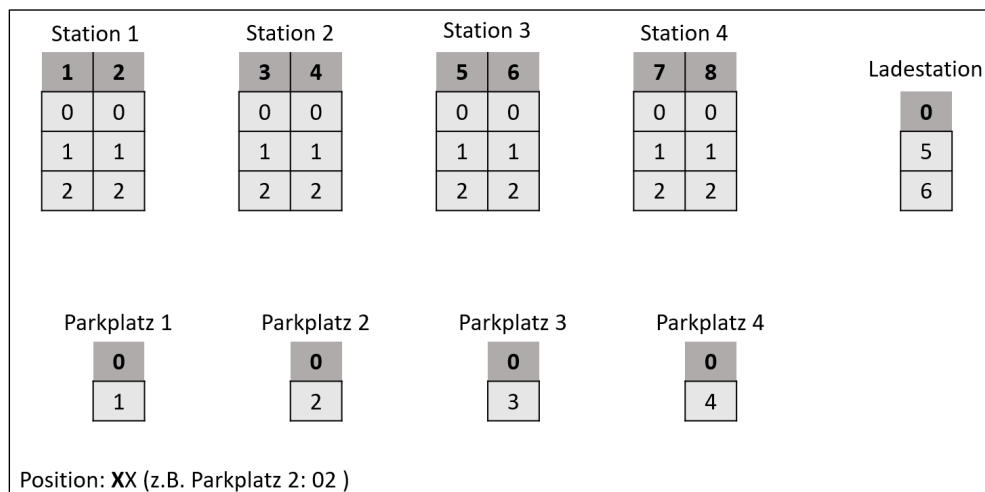


Abbildung 4.2.: Positionskodierung

Jede Position ist kodiert in einer Zahl bestehend aus zwei Ziffern. Die erste Ziffer beschreibt dabei die Stationsnummer aufsteigend von 1 bis 8. Parkplätze und Ladestationen haben als Kennung in der ersten Ziffer eine 0.

Mit der zweiten Ziffer kann die genaue Position innerhalb einer Station spezifiziert werden. In den Stationen 1 bis 8 ist der RFID-Lesekopf mit Ziffer 0 und

#### 4. Schnittstellen

die Arbeitsplätze mit den Ziffern 1 und 2 kodiert. Die Parkplätze sind aufsteigend mit den Ziffern 1 bis 4 durchnummeriert. Hieran anschließend sind beide Ladestationen mit Ziffer 5 und 6 kodiert.

So ergibt sich für Beispielsweise Parkplatz 2 eine Kodierung von 02, für die untere Ladestation die Kodierung 06 oder für den oberen Arbeitsplatz an Station 3 die Kodierung 31.

#### Telegramm vom Robotino zum Fertigungsplanungsrechner

Das zyklisch gesendete Telegramm vom Robotino enthält alle Informationen, die zur Auftragsgenerierung und Robotinoanzeige nötig sind. Alle Telegrammeinträge sind in Tabelle 4.3 dargestellt. Das Telegramm enthält neun Einträge, welche je einen Double-Wert enthalten.

Byte	Inhalt	Beschreibung
1	Error	Errortyp
2	Akku	Prozentwert als Integer
3	Hindernis	Hindernistyp zwischen 00 und 03
4	*Reserved	interner Roboterstatus
5	AliveAbfrage	1: Roboterabfrage erfordert Antwort; 0: egal
6	Status	$X_1X_2X_3$ ; 000: nichts zu tun
7	Greifer	0:leer; 1:voll
8	*Reserved	
9	*Reserved	

Tabelle 4.3.: Telegramm von Gewerk 2

Im ersten Byte sendet der Robotino einen Error. Die einzelnen Errortypen und die nötige Reaktion sind in Kapitel 4.1.3 dargestellt.

Aus dem zweiten Byte kann der aktuelle Akkustand des Robotinos gelesen werden. Dieser liegt zwischen 0, vollständig entladen, und 100, vollständig geladen.

Über das dritte Byte kann gelesen werden, ob der Robotino ein Hindernis erkannt hat. Beim Empfangen einer 00 ist kein Hindernis im Weg und der Robotino kann sich normal bewegen. Eine Änderung des Wertes kann zunächst nur zu 03 erfolgen. Das bedeutet, der Robotino hat ein Hindernis erkannt, dieses aber noch nicht klassifiziert hat. Über die Zahl 01 wird kodiert, dass der Roboter ein Hindernis erkannt hat, dies aber nicht weiter Klassifizieren kann. Mit einer 02 wird angezeigt, dass das erkannte Hindernis ein anderer Robotino ist. Aufgrund der Umgebungsstörungen kann es sein, dass der Robotino immer wieder abwechselnd eine 00 und eine 01 sendet.

Das vierte Byte wird von Gewerk 2 für eine interne Bekanntmachung verschiedener Roboterstati genutzt. Der Wert wird nicht versendet und bleibt immer 0.

Über das fünfte Byte kann der Robotino eine Anfrage triggern. Sobald eine 1 empfangen wird, so wird im nächsten Telegramm an den Robotino der AliveStatus

#### 4. Schnittstellen

zu eins gesetzt. Dadurch kann der Robotino validieren, dass die Kommunikation noch vorhanden ist. Ansonsten ist der Wert 0.

Mit dem sechsten Byte wird der aktuelle Roboterstatus kodiert. Dieser Wert ist für die Auftragsplanung am wichtigsten. Die Zahl besteht immer aus drei Ziffern. Die erste Ziffer dient dabei als Typisierung des Status. Wenn der Status 000 ist, bedeutet das, der Robotino hat nichts zu tun und benötigt einen Auftrag. Auf einem Parkplatz wird keine 000 gesendet. Immer wenn ein Robotino neu eingesetzt wird, seinen aktuellen Transportauftrag abgeschlossen hat, aus einem Errorstatus kommt oder den Ladevorgang an der Lasestation abgeschlossen hat wird eine 000 gesendet.

Wenn die erste Ziffer eine 1 ist, befindet sich der Robotino auf dem Weg zu einer Position. Bei einer 2 als erste Ziffer ist der Robotino gerade an einer bestimmten Position. Über den Wechsel von 1 auf 2 kann somit das Erreichen einer Position erkannt werden oder der Verlassen einer Position über den Wechsel auf 1.

Die beiden letzten Ziffern des Status kodieren die Position in der in Abschnitt 4.1.2 beschriebenen Art.

Das der Greifer des Robotinos geschlossen, also voll, ist wird im siebten Byte mit einer 1 angezeigt. Bei offenem Greifer ist der Wert 0.

Das achte und neunte Byte sind für einfache Erweiterung des Telegramms um weitere Informationen geplant gewesen. Bei Projektstart war das vierte und fünfte Byte ebenfalls reserviert, wurde aber während des Projekts zugewiesen.

##### 4.1.3. Fehlertypen und Behebung

**Fehlertyp 1**

**Fehlertyp 2**

**Fehlertyp 3**

# 5. Implementierung

Implementierung

## 5.1. Programmstruktur

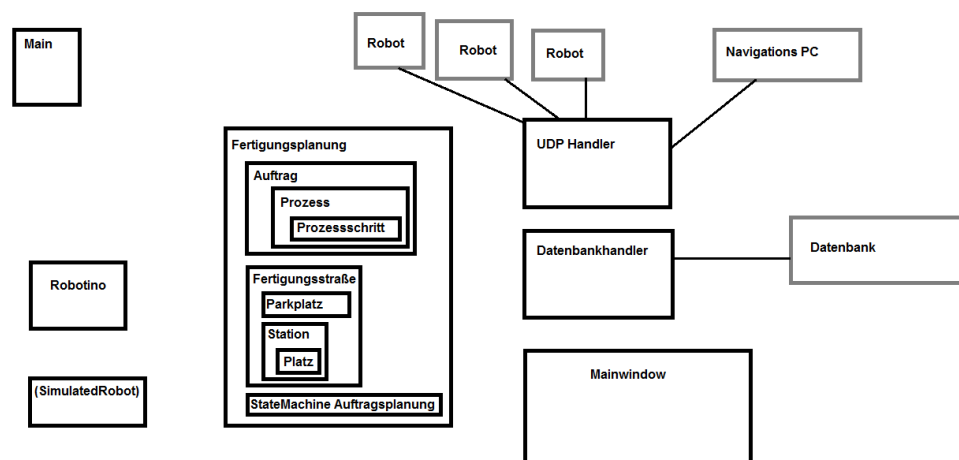


Abbildung 5.1.: Klassendiagramm

## 5.2. Database-Handler

Über die Klasse DatabaseHandler kann eine Verbindung zu der Datenbank auf dem Fertigungsrechner aufgebaut werden und bietet anschließend einen Funktionspool als definierte Schnittstelle zur Datenbank.

### 5.2.1. Konstruktor, Initialisierung und Sendefunktion

Neben dem Konstruktor für die Initialisierung der Datenbankschnittstelle gibt es eine Funktion zur Initialisierung der Datenbank und eine in allen anderen Funktionen aufgerufene Sendefunktion.

### Konstruktor

Beim Aufruf des Konstruktors, siehe Listing 5.1 des Datenbank-Handler wird zunächst eine QSqlDatabase (siehe Kapitel 2.1.3) erzeugt. Dazu wird zunächst in Zeile 1 eine Datenbank vom Typ MySql erzeugt. Um eine Verbindung zur vorhandenen Datenbank herzustellen wird in Zeile 2 - 6 die IP-Adresse mit Port, der Datenbankname und eine Zugangsberechtigung mit Benutzername und Passwort angegeben.

Mit der Funktion open() in Zeile 8 wird versucht eine Verbindung aufzubauen. Über den Rückgabewert der Funktion kann überprüft werden, ob die Verbindung erfolgreich war. Sofern es trotz erfolgreicher Verbindung einen Fehler gibt, beispielsweise fehlende Zugriffsberechtigungen oder anderes, kann dieser wie in Zeile 10 ff dargestellt abgefangen und dargestellt werden.

Listing 5.1: Datenbank-Handler Konstruktor

```
db = QSqlDatabase::addDatabase("QMYSQL");
2 db.setHostName("192.168.0.107");
  db.setPort(3306);
4 db.setDatabaseName("vpj");
  db.setUserName("FP");
6 db.setPassword("1234");

8 bool ok = db.open();

10 QSqlError err;
  if (err.type() != QSqlError::NoError){
12     qCritical() << err.text();
  }
```

### Initialisierung

Um die Datenbank zu Programmstart auf einen definierten Anfangszustand zu bringen wurde eine Initialisierungsfunktion (Listing 5.2) geschrieben. Darin werden alle Zeitstempel, Aufträge und Prozesse (Werkstücke) gelöscht.

Weiterhin wird der Roboter in der Datenbank initialisiert mit Status 0 und ein zugeordnetes Werkstück gelöscht.

Der Parkplatz und Arbeitsplatz wird zurückgesetzt. Dabei wird der Status auf 0 gesetzt und ein zugeordneter Roboter gelöscht. Im Arbeitsplatz wird zusätzlich ein zugeordnetes Werkstück gelöscht.

Der Eintrag des Werkstücks in der Taggen-Tabelle wird zurückgesetzt.

Listing 5.2: Datenbank-Handler Initialisierung

```
SendOverQuery(query, "DELETE FROM vpj.rfid_timestamp;");
2 SendOverQuery(query, "UPDATE vpj.roboter SET
    betriebsstatus=0, Werkstueck_RFID_Werkstueck=NULL;");
```



## 5. Implementierung

```
SendOverQuery(query, "UPDATE vpj.parkplatz SET Status=0,  
    Roboter_id_Roboter=NULL;");  
4 SendOverQuery(query, "UPDATE vpj.arbeitsplatz SET Status  
    =0, Roboter_id_Roboter=NULL, Werkstueck_RFID_Werkstueck  
    =NULL;");  
SendOverQuery(query, "UPDATE vpj.taggen SET  
    Werkstueck_RFID_Werkstueck=NULL WHERE id_taggen=1;");  
6 SendOverQuery(query, "DELETE FROM vpj.werkstueck;");  
SendOverQuery(query, "DELETE FROM vpj.auftrag;");
```

### Allgemeine Sendefunktion

Jede Kommunikation mit der Datenbank beginnt mit der Erzeugung einer Query, in Listing 5.3 Zeile 1 dargestellt. Da dieser Aufruf vor jeder Kommunikation geschieht, wird dieser in allen folgend beschriebenen Funktionen der Übersicht wegen weggelassen, wäre aber immer in der ersten Zeile zu finden.

In der Funktion SendOverQuery() werden die Zeilen 3 und 4 ausgeführt. Dabei wird der String, der den Befehl enthält, in die Query geladen und nachfolgend mit execute() versendet bzw. auf der Datenbank ausgeführt.

Listing 5.3: Datenbank-Handler Senden über Query

```
QSqlQuery query(db);  
  
q.prepare(name);  
q.exec();
```

### 5.2.2. Getter



### 5.2.3. Updates



### 5.2.4. Setter



## 5.3. UDP-Handler

Der UDP-Handler ist für die Datenkommunikation zwischen dem Fertigungsplanungsrechner und dem Robotino als auch dem Navigationsrechner zuständig. Mittels überladenem Konstruktor kann entweder eine Verbindung zu einem Robotino oder eine Verbindung zum Navigationsrechner aufgebaut werden.

### 5.3.1. UDP-Handler für Navigationsrechner

Um auf Daten des Navigationsrechners zu reagieren wird im Konstruktor des UDP-Handler (vgl. Listing 5.4) ein `QUdpSocket` (vgl. Kapitel 2.1.2) erstellt. Der Socket „horcht“ an dem angegebenen Port auf Daten vom Navigationsrechner. Durch das in Zeile 5 verknüpfte `readyRead()`-Signal wird die Funktion `readPendingNavigationData()` aufgerufen. Dieses Signal wird intern vom Socket emittiert, sobald eine Nachricht am spezifizierten Port anliegt.

Listing 5.4: UDP-Handler Konstruktor für Navigationsdatenaustausch

```

UdpHandler::UdpHandler( quint16 port , QObject *parent ) :
    QObject( parent )
2 {
    socket = new QUdpSocket( this );
4    socket->bind( port , QUdpSocket::ShareAddress );
    connect( socket , SIGNAL( readyRead() ) , this , SLOT(
        readPendingNavigationData() ) );
6    lastSendValues = QVector<double>(5);
}

```

Die Funktion `readPendingNavigationData()`, dargestellt in Listing 5.5, liest den aus dem Socket kommenden Datenstrom aus. Solange neue Daten am Socket anliegen werden diese empfangen, in ein Datagramm geschrieben und anschließend in der Funktion `processTheNavigationData()` weiterverarbeitet.

Listing 5.5: Daten über UDP empfangen

```

while ( socket->hasPendingDatagrams() ) {
2    QNetworkDatagram datagram = socket->receiveDatagram();
    processTheNavigationData( datagram );
4 }

```

Das empfangene Datagramm wird zunächst in der Funktion `processTheNavigationData` (Listing 5.6 Zeile 1 - 14) auf die korrekte Länge überprüft. So kann weitgehend ausgeschlossen werden, falsche Daten über den Port zu empfangen. Die Länge von 448 (Zeile 1) ergibt sich aus der Länge der gesendeten Werte multipliziert mit der Länge eines Bytes  $56 * 8 = 448$ .

Wenn die Datagrammlänge korrekt ist, wird ein Vektor mit entsprechend vielen Elementen erzeugt. Dieser Vektor wird elementweise in Zeile 8 - 12 befüllt und anschließend an die Funktion `prepareNavigationDataforRobot()` (Listing 5.6 Zeile 15ff) übergeben.

## 5. Implementierung

Die Rohdaten aus dem Datagramm müssen für jedes Element dekodiert werden. Dazu wird in Zeile 10 aus dem Datagramm ein ByteArray der Länge 8 kopiert, welches das Element *i* beinhaltet. Das erste Element liegt im Datagramm vom ersten bis zum achten Byte vor, das zweite vom 9. bis zum 17. usw. Die so entstehenden ByteArrays werden in Zeile 11 interpretiert als ein Element vom Typ Double.

Listing 5.6: UDP-Handler Navigationsdatenaustausch

```

1  if (datagram.data().length() != 448)
2  {
    qDebug() << "Received Navigation Data have wrong
        dimensions: " << datagram.data().length();
4  }
    else
6  {
        QVector<double> vektor(56);
8        for(int i = 0; i < 56; i++)
        {
10           QByteArray temp = datagram.data().mid(i*8,8);
            vektor[i] = *reinterpret_cast<const double*>(temp.
                data());
12        }
        prepareNavigationDataforRobot(vektor);
14    }
16    QVector<int> posr1(3);
18    posr1[0] = static_cast<int>(values[0]);
    posr1[1] = static_cast<int>(values[1]);
20    posr1[2] = static_cast<int>(qRadiansToDegrees(values[2]));
    [...]
22    emit NavigationDataReceivedR1(posr1);
```

Die entstandenen Doublewerten im übergebenen Vektor werden in der Funktion `prepareNavigationDataforRobot()` ausgewertet. Zeile 15 ff zeigt dies exemplarisch für den ersten Robotino. Je Robotino wird ein neuer Vektor von 3 Elementen erzeugt, der anschließend mit dem X-, Y-Wert und Winkel in Grad befüllt wird. Über ein Signal wird zuletzt dieser Vektor bekanntgemacht.

### 5.3.2. UDP-Handler für Robotino

Der überladene Konstruktor um die UDP-Kommunikation zu einem Robotino aufzubauen ist in Listing 5.7 dargestellt. Anders als beim Navigationsrechner werden zwei Ports und eine IP-Adresse übergeben. Zudem eine Flag, die anzeigt, ob die Kommunikation nur simuliert werden soll um keinen realen Robotino, sondern einen Simulierten (siehe Kapitel 7) einzubinden.

## 5. Implementierung

Im Simulationsmodus wird die State-Machine des simulierten Roboters gestartet und die Sendefunktion des Robotinos mit der Funktion `simulatedRobotDataReceived()` verbunden.

Im Betrieb mit einem real vorhandenen Robotino wird ähnlich dem Navigationskonstruktor der `QUdpSocket` an den Receive-Port gebunden und der Nachrichtenempfang mit der Funktion `readPendingRobotData()` verknüpft.

Listing 5.7: UDP-Handler Konstruktor für Navigationsdatenaustausch

```
UdpHandler::UdpHandler(bool simulated, QHostAddress ip,
    quint16 sport, quint16 rport, QObject *parent) :
    QObject(parent)
2 {
    this->ip = ip;
4    this->receivingport = rport;
    this->sendingport = sport;
6    this->simulated = simulated;
    lastSendValues = QVector<double>(5);
8
    if (!simulated)
10 {
        socket = new QUdpSocket(this);
12        socket->bind(rport, QUdpSocket::ShareAddress);
        connect(socket, SIGNAL(readyRead()), this, SLOT(
            readPendingRobotData()));
14    }
    else
16 {
        r.Start();
18        QObject::connect(&r, &SimulatedRobot::
            StatusDataSimulatedReceived, this, &UdpHandler::
            simulatedRobotDataReceived);
20    }
}
```

Zusätzlich zu den Lesefunktionen, die Daten über UDP empfangen, gibt es für die Robotinos eine Schreibfunktion `WriteData()` (Listing 5.8), die ermöglicht über UDP eine Nachricht an den Robotino zu senden. Dabei werden zunächst die zu sendenden Werte zwischengespeichert. Im Simulationsmodus werden anstelle einer echten Datenübertragung die zu versendenden Daten direkt in die State-Machine des simulierten Roboters geschrieben (Zeile 2-7).

Eine tatsächliche Übertragung (Zeile 9ff) erfordert das Verpacken der Daten in ein Datagramm. Dazu wird ein Byte Array erzeugt, an das die Daten angehängt werden. In Zeile 13 werden die als Double vorliegenden Daten neu interpretiert als Pointer auf einen Char, welche mit der Funktion `fromRawData` als ein Datagramm zurückgegeben werden.

Das Datagramm wird abschließend über den UDP-Socket mit der Funktion `writeDatagram` (Zeile 15) an den angegebenen Roboter über den Sende-Port ge-

schickt.

Listing 5.8: Daten über UDP verschicken

```
lastSendValues = values;
2 if (simulated)
{
4     r.auftragsart = static_cast<int>(values[0]);
    r.pos1 = static_cast<int>(values[1]);
6     r.pos2 = static_cast<int>(values[2]);
}
8 else
{
10     QByteArray Data;
    for(int i = 0; i < 5; i++)
12     {
        Data.append(QByteArray::fromRawData(
            reinterpret_cast<char*>(&values[i]), sizeof(
                values[i]));
14     }
    socket->writeDatagram(Data, ip, sendingport);
16 }
```

Die Funktion `readPendingRobotData` gleicht der bereits im UDP-Handler für Navigationsdaten beschriebenen Funktion (Listing 5.5).

Die nachfolgend aufgerufene Funktion `processTheRobotData` ist bis auf die Längen der Vektoren und des Datagramms gleich der zuvor in Listing 5.6. Die valide Datagrammlänge beträgt bei den Robotern 72, was sich aus Bytelänge multipliziert mit Datenarraylänge  $8 * 9 = 72$  ergibt. Die Auswertung des neun Elementen langen Vektors aus Doublewerten erfolgt erst in der Robotinoklasse, welche auf das Signal `StatusDataReceived` reagiert. Dasselbe Signal wird auch vom simulierten Roboter gesendet, wodurch die Schnittstelle im Simulationsmodus nicht weiter angepasst werden muss.

### 5.4. Robotino

Die real existierenden Robotinos werden in der Klasse `Robotino` abgebildet. In der Klasse werden alle Daten gespeichert, die den Robotino beschreiben. Dazu gehören unter anderem die Position in x, y, der Winkel, eine eindeutige RoboterID. Weiterhin werden alle vom Robotino gesendeten Daten wie Status, Position, Error, Hindernis, Akku und Greifer gespeichert (vgl. 4.1).

Über ein `isAlive`-Tag kann validiert werden, dass der Robotino noch erreichbar ist. Das Defekt-Tag wird von Visualisierung und Auftragsplanung berücksichtigt, um den Robotino händisch außer Betrieb zu schalten.

Es werden demnach deutlich mehr Daten über den Robotino gespeichert, als in der Datenbank hinterlegt werden. Dies kommt auch daher, dass über eine Änderung

## 5. Implementierung

von manchen Werten ein Signal emittiert wird um über ein Event darauf zu reagieren. Diese Funktionalität ist über die Datenbank nicht gegeben.

Beim Erzeugen einer Instanz eines Robotinos wird im Konstruktor sowohl die RoboterID festgeschrieben, als auch die IP-Adresse mit dem zugehörigen Sendeport und Empfangsport. Außerdem werden die Timer mit den zugehörigen Timeout-Funktionen verknüpft.

### 5.4.1. Werte aktualisieren

Wenn der zugehörige UDP-Handler eines Robotinos neue Werte bereitstellt (vgl. `sec:UdpHandler`) wird die Funktion `UpdateValues()` aufgerufen. Bei jedem Aufruf der Funktion wird der Robotino als Alive markiert und dies bei einer Änderung als Signal emittiert. Zusätzlich wird der 30 Sekunden laufende `AliveTimer` gestartet oder zurückgesetzt.

Wenn sich der anliegende `Robotererror` geändert hat wird dieser kategorisiert und den bekannten `Robotererror`typen (vgl. 4.1) zugeordnet. Folgend wird ein Signal emittiert, welches eine Änderung des Errors anzeigt und den Errortyp beinhaltet.

Bei einer Änderung von Greifer, Akku, Hindernis oder `UDPAlive` wird jeweils der Wert gespeichert und die Änderung über ein spezifisches Signal emittiert.

Eine Statusänderung des Robotinos bewirkt eine Vorauswertung des empfangenen Statuswerts. Der zusammengesetzte Status wird wieder in seine drei Bestandteile zerlegt und über ein Signal bekanntgemacht. Bei einem Status von über 200 wird die Roboterposition ebenfalls gespeichert.

### 5.4.2. Visualisierungsfunktionen

Bei Empfangen von Navigationsdaten vom Navigationsrechner zu einem Robotino wird in der Funktion `UpdatePosition()` die Roboterposition angepasst. Dazu wird zunächst überprüft, ob sich die Position um mehr als 2 in X- und Y-Richtung und um mehr als 1 im Winkel geändert hat. Damit wird eine Glättung der Roboterposition erzeugt und ein „zittern“ in der Visualisierung aufgrund der schwankenden Position unterdrückt.

Der Sonderfall, dass alle drei empfangenen Werte 0 sind bedeutet, dass der Navigationsrechner den Robotino verloren hat. In diesem Fall wird die Roboterposition nicht aktualisiert und stattdessen ein `NavigationsTimer` gestartet. Wenn nach fünf Sekunden keine neue Robotinoposition erkannt wird, so wird der Robotino in der Visualisierung verdeckt. Dadurch wird eine kurzzeitige Verdeckung durch z.B. die Kabelgänge an den Stationen der Robotino immer noch angezeigt und verschwindet nicht.

Sobald sich der `Alive`-Tag ändert wird in der Visualisierung die `RoboterAlive`-LED angepasst. Ein Roboter wird als Alive gewertet bei Aufruf der Funktion `UpdateValues()`. Wenn der `AliveTimer` abläuft wird der Robotino als nicht Alive

gewertet. Das führt dazu, dass ein Robotino auch als lebend angezeigt wird, wenn die Kommunikation nicht vollständig funktioniert, solange dieser seinen Status an den Fertigungsplanungsrechner sendet. Nach maximal 30 Sekunden wird ein Absturz der Kommunikation erkannt und der Robotino wird nicht länger in der Auftragsvergabe berücksichtigt.

### 5.5. Auftrag, Prozess, Prozessschritt

Die im Programm auftretenden Aufträge sind in einer bestimmten Struktur organisiert. Diese variiert zu der in der Datenbank abgespeicherten Struktur. So wird über die Visualisierung ein Auftrag angelegt. Jeder so angelegte Auftrag beinhaltet ein oder mehrere Prozesse. In der Datenbank sind dies komplementär gesehen die Werkstücke. Jeder Prozess besteht aus mehreren Prozessschritten, die den genauen Ablauf eines Prozesses beschreiben.

Auftrag, Prozess und Prozessschritt sind in verschiedene Klassen geschrieben. Folgend werden diese Klassen näher beleuchtet.

#### 5.5.1. Prozessschritt

Ein Prozessschritt besteht aus genau einer Stations-ID verknüpft mit einer festen Bearbeitungsdauer in Sekunden. Dies beschreibt für einen Teil eines Prozesses. In einer Fortschritts-Flag wird festgehalten, ob der Prozessschritt bereits bearbeitet wurde, oder noch bearbeitet werden muss.

Der Prozessfortschritt kann ein Signal emittieren, welches anzeigt, dass sich der Fortschritt geändert hat.

#### 5.5.2. Prozess

Ein Prozess, in der Datenbank Werkstück, beinhaltet eine Liste von Prozessschritten. Diese Liste bestimmt mit ihrer Reihenfolge und Länge die Bearbeitungsvorschrift für ein Werkstück. Jeder Prozess besitzt zudem einen Fortschritt, eine eindeutige ID und eine ReferenzID.

Mit der eindeutigen ID kann später ein Prozess innerhalb des Programms wiedergefunden werden. Nur die in Aufträgen befindlichen Prozesse erhalten eine aufsteigende laufende Nummer als eindeutige ID. Die in der Initialisierung erzeugten Prozesse oder die über die Visualisierung erzeugten Prozesse (folgend Referenzprozesse) erhalten keine spezielle eindeutige ID, sondern eine aufsteigende eindeutige ReferenzID. Dies führt dazu, nicht alle Informationen in den einzelnen Prozessen halten zu müssen, sondern auf den initialisierten Referenzprozess zugreifen zu können.

Die ReferenzID eines Prozesses enthält die eindeutige ID des zugehörigen Referenzprozesses.

## 5. Implementierung

Zur Prozesskontrolle gibt es zusätzlich noch verschiedene Flags die den Status des Prozesses widerspiegeln. Dazu gehört ein Blocked- und ein InProgress-Flag. Das InProgress-Flag wird gesetzt, sobald ein Werkstück an einer Station bearbeitet wird. Das Blocked-Flag kann über die Hard-Code Area (vgl. Kapitel 6.5.4) gesetzt werden und verhindert eine weitere Bearbeitung.

Jeder Prozess enthält einen Timer, der die verbrauchte Zeit an einem Arbeitsplatz kontrolliert. Der Timer wird gestartet, sobald ein Werkstück an einer Station bearbeitet wird, die nötige Zeit wird dem zugehörigen Prozessschritt entnommen. Bei Ablauf des Timers wird die Funktion `TimerElapsed()` aufgerufen, in der der zugehörige Prozessschrittfortschritt auf fertig gesetzt und das InProgress-Flag zurückgesetzt wird.

Über die beiden Signale `ProzessFinished` und `FortschrittUpdated` kann bekannt gemacht werden, dass sich ein Fortschritt eines Prozesses geändert hat oder die Arbeit an einer Station beendet wurde und das Werkstück bereit ist für Abholung.

Mit der im Prozess enthaltenen Funktion `UpdateFortschritt()` wird der Fortschritt eines Prozesses anhand der beinhalteten Prozessschritte aktualisiert. Dazu wird die Prozessschrittliste durchgegangen und für jeden Prozessschritt, der als fertig markiert wurde ein entsprechender Fortschritt zum Prozessfortschritt addiert. Die einzelnen Prozessschritte sind dabei so gewichtet, dass die Summe aus allen Prozessschritten, wenn alle abgeschlossen sind, 100 ergibt. Durch Rundungsfehler können in der aktuellen Umsetzung nicht mehr als 25 Prozessschritte je Prozess sauber dargestellt werden. Sollten mehr Prozessschritte erforderlich sein, so müsste die Fortschrittsberechnung angepasst werden.

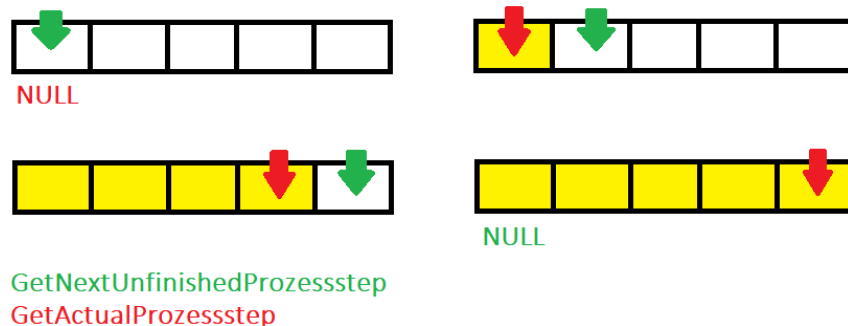


Abbildung 5.2.: Visualisierung des Rückgabewertes der Funktionen `GetNextUnfinishedProzessstep()` und `GetActualProzessstep()`

Um auf den richtigen Prozessschritt innerhalb der Liste zuzugreifen sind zwei unterschiedliche Funktionen implementiert. Beide Funktionen unterscheiden sich ausschließlich in ihrem Rückgabewert. In Abbildung 5.2 sind alle möglichen Fälle Prozessschrittliste leer oben links, ein oder mehrere Prozessschritte in der Liste oben rechts und unten links und Liste voll unten rechts dargestellt. Dabei ist zu erkennen, dass die Funktion `GetNextUnfinishedProzessstep()` immer den nächsten Prozessschritt zurückliefert, der noch nicht bearbeitet wurde. Bei voller Liste wird `NULL` zurückgegeben. Anders hierzu die Funktion `GetActualProzessstep()`, die `NULL` bei leerer Liste zurückgibt und sonst den letzten fertigen Prozessschritt.



## 5. Implementierung

Die Funktion `GetActualProzessstep()` wird in der Fertigungsplanung zur Planung der Aufträge verwendet (siehe 5.6.5). `GetNextUnfinishedProzessstep()` findet Anwendung sobald auf einem aktuellen Prozess eine Änderung auftritt oder umgesetzt werden soll. Wenn beispielsweise der Timer abgelaufen ist, wird in der Funktion `TimerElapsed()` über der zurückgegebene Prozessschritt als fertig markiert.

### 5.5.3. Auftrag

Die Klasse `Auftrag` enthält eine Liste von Prozessen und darin enthaltenen Prozessschritten und bündelt somit alle drei Klassen. Ähnlich dem Prozess gibt es einen Fortschritt, der sich anteilig aus den einzelnen Prozessfortschritten errechnet und eine eindeutige ID mit derer der Auftrag wiedergefunden und verwaltet werden kann.

Mit dem Signal `FortschrittUpdated` kann bekannt gemacht werden, dass sich der Fortschritt geändert hat und z.B. die Visualisierung aktualisiert werden. Das Signal wird gesendet, sobald die Funktion `UpdateFortschritt()` aufgerufen wurde. In dieser Funktion werden alle enthaltenen Prozesse auf ihren Fortschritt überprüft und im Auftrag gegebenenfalls angepasst. Die Fertigungsplanung verknüpft Auftrag, Prozess und Prozessschritt derartig, dass ein Fortschrittsupdate innerhalb eines Prozessschrittes eine Aktualisierung des Prozessfortschritts hervorruft. Dies wiederum lässt den entsprechenden Auftragsfortschritt aktualisieren (siehe Listing 5.9).

Ein bestimmter Prozess aus der Liste kann über die Funktion `GetProzessByID()` mit seiner eindeutige ID zurückgegeben werden.

Weiterhin werden zwei Funktionen bereitgestellt, die von der Fertigungsplanung genutzt werden um Prozesse oder Prozessschritte zu erhalten. Die Funktion `GetNextUnfinishedProzesssteps()` gibt, mit Hilfe der Funktion `GetNextUnfinishedProzessstep()` aus dem Prozess, eine Liste aller entsprechenden Prozessschritte zurück (vgl. Abschnitt 5.5.2). Über die Funktion `GetAllUnfinishedProzesses()` werden alle Prozesse zurückgegeben, die noch unfertige Prozessschritte enthalten, sofern diese nicht blockiert sind. Hiermit werden in der Fertigungsplanung die zu bearbeitenden Prozesse ausgewählt (siehe 5.6.5).

## 5.6. Fertigungsplanung

Die Klasse `Fertigungsplanung` enthält alle verfügbaren Roboter, die Datenbank-schnittstelle, die abzuarbeitenden Aufträge (vgl. Abschnitt 5.5), die Fertigungsstraße und eine State Machine zur Verteilung von Roboteraufträgen. Außerdem wird auf Events reagiert, die von der Visualisierung oder anderen Quellen kommen. Zu den Events gehören Statusänderungen des Roboters, hinzufügen von Aufträgen oder Prozessen und Funktionen, die über die Hard-Code Area (vgl.

Kapitel 6.5.4) angestoßen werden können. In der Fertigungsplanungsklasse werden somit alle Funktionalitäten gebündelt, die nicht direkt mit der Visualisierung zusammenhängen.

### 5.6.1. Initialisierung

Um die Fertigungsplanung zu initialisieren wird zunächst in der Funktion `InitProzesses()` ein Grundzustand hergestellt, in der mindestens 5 Prozesse vorhanden sind. Dazu werden zunächst über einen Datenbankaufruf alle dort verfügbaren Prozesse in eine Liste geschrieben.

Sollte die Liste weniger als 5 Elemente beinhalten, also in der Datenbank zu Programmstart weniger als 5 Prozesse vorhanden sein, so werden 5 neue zuvor definierte Prozesse (Abschnitt 5.6.1) angelegt und die Liste so überschrieben.

Die erzeugten Prozesse werden abschließend in die Datenbank geschrieben. Außerdem erhalten alle Prozesse gemäß ihrer Schritte und Dauer in der Visualisierung einen entsprechenden Tooltip (vgl. Kapitel 6.9.1).

Die Prozessliste kann über die Funktion `AddProzess()` dynamisch während das Programm läuft erweitert werden. Dies geschieht bei absenden eines eingestellten Prozesses in der Visualisierung (6.7).

### Aufträge hinzufügen

Durch Einstellen und Absenden eines Auftrags in der Visualisierung (6.8) wird die Funktion `AddAuftrag()` aufgerufen. In dieser Funktion wird zunächst ein leeres Auftragsobjekt erzeugt und diesem die nächst höhere Auftragsnummer aus der Datenbank zugeordnet.

Für jeden Prozesseintrag aus der Visualisierung wird nun die angegebene Anzahl des jeweiligen Prozesses eine Prozesskopie erzeugt. Den so kopierten Prozessen werden alle zugehörigen Prozessschritte als Kopie angehängt, um eigenständige Aufträge zu erhalten. Zuletzt werden die Prozesse und Aufträge miteinander verknüpft (siehe Listing 5.9).

Listing 5.9: Auftragserzeugung und Verknüpfungen

```
Auftrag *A = new Auftrag();  
2  
for (int prozesscounter = 0; prozesscounter < 5;  
    prozesscounter++) //5 Prozess UI Elemente  
4 {  
    for (int i = 0; i < Avalues[prozesscounter]; i++) //  
        Anzahl der eingegebenen Prozesszahl  
6    {  
        Prozess *p = new Prozess();  
8        [...] // Prozess + Prozessschritte kopieren  
        A->Prozesse.append(p);
```

## 5. Implementierung

```
10      QObject::connect(p, &Prozess::FortschrittUpdated ,  
                        A, &Auftrag::UpdateFortschritt);  
12      QObject::connect(p, &Prozess::ProzessFinished ,  
                        this , &Fertigungsplanung::StationReady);  
    }  
}
```

Abschließend wird der Auftrag der Auftragsliste angehängt, an die Datenbank gesendet und ein Auftragsitem in der Visualisierung erzeugt (Kapitel 6.4).

### Standardprozesse

Folgend werden die in Abschnitt 5.6.1 erwähnten Standardprozesse näher beleuchtet.

Um ein möglichst ausgeglichenes Fertigungsbild zu schaffen wurden die fünf Standardprozesse so untereinander abgestimmt, dass diese unterschiedlichen Kriterien genügen.

Zunächst wurde darauf geachtet, dass die Prozesse unterschiedlich viele Prozessschritte beinhalten. Abbildung 5.3 kann entnommen werden, dass die Prozesse zwischen zwei und sechs einzelne Prozessschritte erfordern. In der Abbildung ist weiterhin dargestellt, welche Station als Start und Ziel je Prozessschritt angefahren wird.

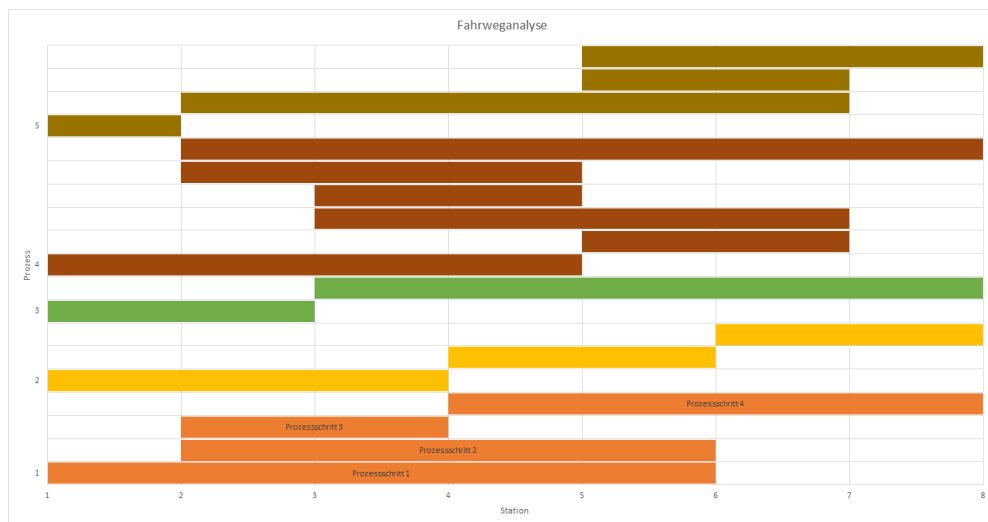


Abbildung 5.3.: Diagramm Fahrweganalyse

Zu erkennen ist, dass jeder Prozess bei Station 1 beginnt und in Station 8 endet. Die Einzelnen Prozesse (siehe farbliche Unterscheidung) sind dabei von unten nach oben zu lesen, beginnend mit Prozessschritt 1.

Aus dem Diagramm können weiterhin potentielle Kollisionsherde und „Staugefahren“ erkannt werden. Die Standardprozesse wurde so angepasst, dass alle Stationen möglichst gleichmäßig häufig durchfahren werden. Weiterhin kann aus dem Diagramm abgelesen werden, ob durch ungünstige Auftragsplanung ein Deadlock

## 5. Implementierung

angefahren werden kann, also alle Aufträge irgendwann durch andere Aufträge blockiert sein können. Da keine Schleifen zwischen den Stationen entstanden sind kann gewährleistet werden, dass alle Aufträge in jeder Belegung nach einiger Zeit abgearbeitet werden können, wenn die Werkstücke in Station 8 entnommen werden.

Das Diagramm in Abbildung 5.4 zeigt eine genauere Analyse der Stationen. Angenommen wird, dass jeder der Standardprozesse genau ein mal abgearbeitet wird. Auf der linken y-Achse, die zu den bunten Säulen gehört, ist die Belegungsdauer jeder Station in Sekunden angegeben. Die rechte, den hellgelben Säulen zugeordnete y-Achse bezeichnet die Anzahl der Anfahrten an eine Station.

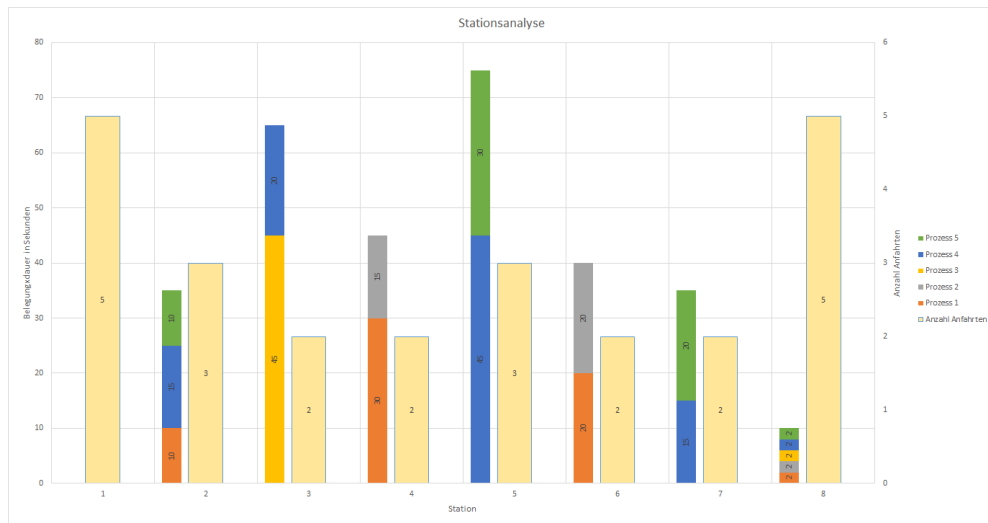


Abbildung 5.4.: Diagramm Stationsanalyse

Es ist zu erkennen, dass Station 1 und 8 genau fünf mal angefahren werden, also je Prozess ein mal. Die weiteren Stationen 2 bis 7 werden zwei- bis dreimal angefahren. So wird sichergestellt, dass die Stationen möglichst gleichmäßig belastet werden.

Die farblich unterschiedlichen Säulen zeigen die kürzeste Belegungsdauer der Arbeitsplätze an. Die Balken ergeben sich durch die Dauer eines einzelnen Prozessschrittes an einem Arbeitsplatz. Es Belegungsdauer für alle Prozesse zwischen 35 und 75 Sekunden je Station liegt. Die 2 Sekunden Bearbeitungszeit je Prozess an Station 8 sind dabei vernachlässigt worden und gewährleisten nur ein sicheres Herausnehmen der Werkstücke durch den Menschen, ohne dass sich der Roboter noch im Entladezyklus befindet.

Als Besonderheit ist an Station 5 zu erkennen, dass es drei Anfahrten, aber nur zwei verschiedene Prozesse gibt. Dies liegt darin begründet, dass Prozess 4 Station 5 zwei mal anfährt mit verschiedenen langen Zeiten. Dadurch ist die Gesamtzeit in Station 5 auch am höchsten, gehört jedoch zu großen Anteilen zum selben Prozess. Station 2 hat die kürzesten Bearbeitungszeiten je Prozess, da diese Station von drei verschiedenen Prozessen angefahren wird. Gegenätzlich dazu ist die Bearbeitungszeit an Station 23 von Prozess 3 mit 45 Sekunden sehr hoch angesetzt, da dieser Prozess keine weitere Station anfährt.

## 5. Implementierung

Es wurde versucht, die durchschnittliche Belegungsdauer der Stationen anzugleichen, um alle Arbeitsplätze möglichst homogen auszulasten. Zu erkennen ist, dass die minimale

Als Nebeneffekt ergibt sich, dass die Roboter möglichst ausgeglichen über die gesamte Fertigungsstraße fahren und die Gefahr von Staus oder Kollisionen dadurch verringert wird.

### 5.6.2. Roboter Statusänderungen

Wenn eine der in der Fertigungsplanung enthalten Roboterklassen ein Event feuert, welches eine Statusänderung beinhaltet (vgl. Kapitel 5.3), wird über Funktionen in der Fertigungsplanung auf diese reagiert. So wird der Greifer oder das Statusflag eines Roboters bei einer Änderung aktualisiert. Beim Empfanges eines Errorevents des Roboter wird je nach Errorotyp eine Meldung ausgegeben und individuell auf die Robotererror reagiert.

Weiterhin wird durch Ablauf des Timers eines Arbeitsplatzes dieser in der Fertigungsstraße aktualisiert und auf bereit geschaltet.

#### Roboterstatus geändert

Bei Änderung des Roboterstatus wird die Funktion `OnRobotStatusChanged()` aufgerufen. Dabei wird zunächst der aufrufende Roboter ausgewählt und zwischengespeichert. Sollte sich der Status des Roboter auf eins geändert haben, also der Roboter auf dem Weg zu etwas ist, so wird unterschieden ob sich der Roboter auf einem Parkplatz oder einem Arbeitsplatz befunden hat.

Von einem Parkplatz (auch Ladestation) kommend wird dieser in der Datenbank und der Visualisierung wieder freigegeben. Wenn der Roboter zuletzt an einer Station war, so wird diese jetzt für andere Roboter freigegeben und der Roboter wird in der Datenbank aus dem Arbeitsplatz gelöscht.

#### Greiferstatus geändert

Die Prozessaktualisierungen basieren auf der Änderung des Greifers des Roboters. Es wird vorausgesetzt, dass sich der Greifer zu keinem Zeitpunkt willkürlich öffnet oder schließt, sondern nur bei Aufnahme oder Abgabe eines Werkstücks an einer Station.

Das Schließen des Greifers eines Roboters bedeutet somit immer, dass ein Werkstück aufgenommen wurde. Somit wird in der Datenbank der Arbeitsplatz, an dem sich der Roboter befindet freigegeben und kann für andere Aufträge genutzt werden. Der Arbeitsplatz wird in der Visualisierung und der Datenbank freigegeben. Zuletzt wird der Tooltip des Arbeitsplatzes (vgl. Kapitel 6.9.1) auf einen leeren String gesetzt.

## 5. Implementierung

Beim Öffnen des Greifers wird ein Werkstück in einen Arbeitsplatz einer Station gelegt. Daraufhin wird die Zuordnung des Werkstücks zu dem entsprechenden Roboter gelöscht. Mit der Werkstücks-ID kann nun aus der Auftragsliste der zugehörige Prozess identifiziert werden. Der Timer des so gefundenen Prozesses wird jetzt gestartet, somit läuft die Bearbeitung des Werkstücks an dem Arbeitsplatz. In der Visualisierung und Datenbank wird der Arbeitsplatz von reserviert auf belegt aktualisiert.

Weiterhin wird das Werkstück, dass dem Roboter zu diesem Zeitpunkt zugeordnet ist an der Station an der sich der Roboter zuletzt aufgehalten hat gelöscht und der zugehörige Arbeitsplatz wird freigegeben.

### Roboter Error geändert

Bei Auftreten eines Errors im Roboter wird je nach Typ eine Error-Meldung ins Log geschrieben.

Die Error-Meldungen des Roboters wurden bewusst nicht automatisiert behandelt, da dem Benutzer die Kontrolle über das Gesamtsystem behalten sollte. Somit muss je nach Fehlertyp eine unterschiedlicher Lösungsweg verfolgt werden.

Die auftretenden Fehler und der Lösungsweg ist in Kapitel 4.1.3 zu finden.

### 5.6.3. Hard-Code Funktionen

Die mit der Hard-Code Area erzeugten Befehle aus der Visualisierung (vgl. Kapitel 6.5.4) werden in verschiedenen Funktionen verarbeitet. So wird beispielsweise der Parkplatz oder eine Station in Datenbank und Visualisierung freigegeben, ein Roboter defekt geschaltet oder ein Arbeitsplatz als defekt markiert oder repariert.

### 5.6.4. Zustandsdiagramm

Das Zustandsdiagramm in Abbildung 5.5 beschreibt die Logik der Auftragsvergabe an die Roboter. Das Diagramm wurde anschließend als State Machine (vgl. 2.1.4) implementiert.

In der Abbildung sind die Zustände mit Namen dargestellt. Alle Zustände haben eine komplementäre Funktion, die nach einem Zustandswechsel als Entry-Action aufgerufen wird. Eine genaue Funktionsbeschreibung der Zustände ist in Kapitel 5.6.5 zu finden. Um zwischen den Zuständen zu wechseln werden die mit Pfeilen eingezeichneten Transitionen verwendet. Die Pfeilbeschriftung beinhaltet dabei das Signal, welches für den Zustandswechsel genutzt wird. Um das Diagramm übersichtlich zu gestalten, wurden die Timeout-Transitionen nicht vollständig eingezeichnet, sondern enden in einem kreisförmigen Xa-Zustand. Dieser führt direkt zu dem XM-Zustand oben links und leitet in den Zustand Timeout weiter.

### 5.6.5. State Machine Implementierung

Im Quellcode werden zunächst alle Zustände erzeugt und der State Machine hinzugefügt. Anschließend werden alle Transitionen zwischen den Zuständen erzeugt und der State Machine hinzugefügt. Hierbei werden definierte Signale genutzt, die zuvor in der Header Datei deklariert wurden. In der Main-Funktion werden abschließend die Zustände mit den entsprechenden zugehörigen Funktionen verknüpft (vgl. 5.8).

Um die State Machine zu starten wird, nachdem der Initialzustand bekannt gemacht wurde, die Start-Funktion aufgerufen. Eine verkürzte Darstellung der Aufrufe ist in Listing 5.10 abgebildet.

Listing 5.10: Start und Initialisierung der State Machine

```
2      QStateMachine StateMachine;  
      QState *StateCheckRobots = new QState;  
  
4      StateMachine.addState(StateCheckRobots);  
      StateCheckRobots->addTransition(this, SIGNAL(nextState  
          ()), StateCheckAkku);  
  
6      StateMachine.setInitialState(StateCheckRobots);  
8      StateMachine.start();  
  
10     QObject::connect(StateCheckRobots, &QState::entered,  
        this, &Fertigungsplanung::CheckRobots);
```

#### Zustand CheckRobots

Im Initialzustand der State Machine, CheckRobots, werden alle vier Roboter nacheinander auf Bereitschaft überprüft. Die Reihenfolge der zu überprüfenden Roboter wird dabei durch einen hierfür entwickelten Shuffle-Algorithmus (5.6.6) bestimmt.

Ein Roboter gilt als bereit, wenn er als Status 0 zurückgibt, nicht als defekt markiert ist und als lebend gilt. Sobald ein geprüfter Roboter alle drei Kriterien erfüllt wird dieser ausgewählt und in den weiteren Zuständen, gefolgt von CheckAkku, genutzt. Die anderen Roboter werden bis zum nächsten Aufruf des Zustands CheckRobots vernachlässigt.

Sollte keiner der vier Roboter alle drei Kriterien erfüllen wird in den Zustand CheckParkplatzRobots gewechselt.

#### Zustand CheckParkplatzRobots

Wie im Zustand CheckRobots werden alle vier Roboter anhand von Kriterien auf Bereitschaft überprüft. Dazu wird ebenfalls die Reihenfolge zufällig bestimmt.

## 5. Implementierung

Anstatt den Roboterstatus wie beim Zustand CheckRobots auf 0 zu prüfen, wird ein Status zwischen 201 und 204 erwartet. Diese vier Stati repräsentieren die vier Parkplätze auf denen sich ein Roboter befinden kann.

Durch die Aufteilung der Zustände CheckRobots und CheckParkplatzRobots erfolgt eine Priorisierung der Roboter, die sich bereits in den Stationen befinden und einen Auftrag fertig abgearbeitet haben. Nur wenn kein Roboter in den Stationen verfügbar ist wird auf die auf den Parkplatz befindlichen Roboter zurückgegriffen. Dadurch werden Situationen vermindert, in denen sich die Roboter bei der Stationsarbeit gegenseitig behindern.

Sollte keiner der auf den Parkplatz befindlichen Roboter alle weiteren Kriterien erfüllen wird in den Zustand Wait gewechselt.

### **Zustand Wait**

Da in Qt aufgrund der eventbasierten Programmierung (vgl. 2.1.1) keine Endlosschleifen vorgesehen sind wurde im Zustand „Wait“ mittels Timer die Zustandsmaschine künstlich verlangsamt. Ein Entfernen des Zustands führt zum Programmabsturz, da in kürzester Zeit die interne Eventliste voll geschrieben wird. Auf externe Events wie Mauseingaben oder Ereignisse vom Betriebssystem könnte somit nicht mehr reagiert werden.

Durch den Zustand Wait können andere Programmevents und Ereignisse des Betriebssystems abgearbeitet werden, während 2000 ms auf das timeout()-Signal des Timers gewartet wird.

Nach Ablauf des Timers wird in den Initialzustand CheckRobots gewechselt.

### **Zustand CheckAkku**

Der Zustand CheckAkku gewährleistet einen Tiefentenladungsschutz der Roboter. Sobald ein Roboter in den ersten beiden Zuständen ausgewählt wurde wird überprüft, ob sein Akkustand größer gleich 25 Prozent ist. Es wird davon ausgegangen, dass kein Auftrag plus eine Fahrt zum Laden mehr als 25 Prozent Akku verbraucht. Der Wert wurde empirisch ermittelt.

Sollte der Akkustand unter 25 Prozent liegen, wird der Roboter über Zustand SendLoadingTask zu einer freien Ladestation geschickt. Wenn keine Ladestation frei ist wird in den Zustand Wait gewechselt. Bei einem Akkustand von über 25 Prozent wird der Zustand CheckAuftrag aufgerufen.

### **Zustände SendLoadingTask, SendParkingTask, SendPositionTask**

Um bei Kommunikation mit den Robotern in den Zuständen SendLoadingTask, SendPositionTask und SendParkingTask sicherzustellen, dass die Nachricht empfangen wurde, wird diese mehrfach gesendet, bis die Übertragung erfolgreich war (siehe Kapitel 4.1.1). Es wird nach jedem Sendevorgang ein kurzer Wartezustand aufgerufen, der, gleich dem Zustand Wait, einen Programmabsturz verhindert.



## 5. Implementierung

Mit dem individuellen Wartezustand kann ebenfalls das bei aktuell 700 ms ange setzte Sendeintervall eingestellt werden.

Aus allen den drei Send.....Task Zuständen und ihren zugehörigen Wartezuständen kann über ein Ablauf eines Timers in den Zustand Timeout gewechselt werden. Der Timer wird beim erstmaligen Eintritt in einen der Zustände Send.....Task gestartet und läuft nach 25 Sekunden ab. Dieser Timer gewährt eine Absturzsicherheit des Programms bei Roboterausfall oder anderweitigem Fehlschlagen der Kommunikation.

In allen drei Send.....Task Zuständen wird ein erfolgreicher Sendevorgang damit erkannt, dass der Roboterstatus nicht mehr 0 ist, und der Roboter sich nicht mehr auf einem Parkplatz befindet. Solange die beiden Bedingungen nicht erfüllt sind wird weiter an den Roboter gesendet.

Wenn im Zustand SendLoadingTask die Nachricht erfolgreich versendet wurde, wird der Timer für das Timeout gestoppt, und die benötigte Ladestation in der Visualisierung und in der Datenbank reserviert. Abschließend wird in den Initialzustand gewechselt.

Der Zustand SendParkingTask verhält sich Analog dem SendLoadingTask, nur wird bei erfolgreicher Nachrichtenversendung der benötigte Parkplatz reserviert.

Bei dem Zustand SendPositionTask, in dem ein Auftrag aus der Planung an den Roboter verschickt wurde, werden nach erfolgreichem Nachrichtenversand mehrere Aktionen durchgeführt. Zunächst wird der Timeout-Timer gestoppt. Sollte der Auftrag den ersten Prozessschritt, also das Abholen eines Werkstücks im Lager an Station 1, beinhalten wird der Schreibkopf des RFID neu beschrieben. In der Datenbank wird außerdem dem Start- und Ziellarbeitsplatz der genutzte Roboter zugeordnet. Am Startarbeitsplatz wird das vorhandene Werkstück in der Datenbank entfernt und dem Ziellarbeitsplatz zugeordnet. Die beiden Stationen werden außerdem Reserviert um zu verhindern, dass ein anderer Roboter an die zu bearbeitenden Stationen geschickt wird. Somit ist eine erste Kollisionsvermeidung implementiert. Zuletzt wird in der Datenbank das genutzte Werkstück dem Roboter zugewiesen. Auch in der Visualisierung werden Start- und Ziellarbeitsplatz reserviert. Schlussendlich wird in der Visualisierung der Tooltip (vgl. Kapitel 6.9.1) des Start- und Ziellarbeitsplatz aktualisiert.

### **Zustand CheckAuftrag**

Der Zustand CheckAuftrag enthält die Vergabe der Aufträge an die Roboter. Dazu wird zunächst in der Auftragsliste jeder Auftrag auf unfertige Prozesse untersucht und diese folgend an eine neu erzeugte Prozessliste angehängt (vgl. Abschnitt 5.5). Jeder Prozess der so erzeugten Prozessliste wird anschließend auf die Möglichkeit der Bearbeitung geprüft. Sofern eine der folgenden Prüfungen fehlschlägt wird der nächste Prozess geprüft.

Dazu wird zunächst überprüft ob die Auftragsplanung pausiert ist oder sich der Prozess schon in Bearbeitung befindet.

## 5. Implementierung

Es wird geprüft ob die Start und Zielstation nicht durch einen Roboter reserviert ist.

Sollte der nächste Prozessschritt der bearbeitet werden muss an der Lagerstation (Station 1) sein, so wird gewährleistet, dass sich ein Werkstück im Lager befindet und dieses ausgewählt.

Es wird geprüft ob eine der beiden Zielarbeitsplätze der Zielstation frei ist, also weder reserviert noch defekt und anschließend ausgewählt.

Wenn alle Vorbedingungen für einen der Prozesse eintreffen wird der Zustand direkt verlassen und der Auftrag wird an den ausgewählten Roboter im Zustand `SendPositionTask` gesendet.

Sollten alle Prozesse zu keinem Sendevorgang geführt haben, das heißt es ist entweder kein Auftrag vorhanden oder das senden wurde blockiert, so wird der aktuell ausgewählte Roboter zum Parken geschickt (Zustand `SendParkingTask`), sofern er sich noch nicht auf einem Parkplatz befindet. Sonst wird in den Initialzustand gewechselt.

### 5.6.6. Shuffle-Algorithmus

Der Shuffle-Algorithmus dient dazu, die Roboter in zufälliger Reihenfolge abzuarbeiten, um eventuelle Dead-Locks zu vermeiden. Ein solcher Dead-Lock könnte zum Beispiel eintreten, wenn Roboter 1 und 2 an der Ladestation sind, Roboter 3 zum laden geschickt werden müsste, jedoch warten muss. Roboter 4 würde dabei, selbst wenn er bereit wäre einen Auftrag abzuarbeiten nie überprüft werden.

Zurückgegeben wird eine Liste in der die Zahlen 1 bis 4 in zufälliger Reihenfolge vorliegen. Der Algorithmus ist optimiert gegenüber der Anzahl an Systemaufrufen für eine Zufallszahl. Andernfalls wäre es einfacher solange eine Zufallszahl zurückgeben zu lassen, wie die Liste diese noch nicht enthält.

Um die Roboterreihenfolge festzulegen werden vier Schritte durchgeführt. Im ersten Schritt wird eine leere Liste erzeugt und eine zufällige Zahl zwischen 1 und 4 an die erste Stelle geschrieben (Listing 5.11 Zeile 1-3).

Listing 5.11: Shuffle-Algorithmus

```
2   QList<int> liste ;
    int i = QRandomGenerator::global()->bounded(1, 5);
    liste.append(i);
4   i = QRandomGenerator::global()->bounded(1, 4);
    (liste.contains(i)) ? liste.append(i+1) : liste.append
        (i);
6   i = QRandomGenerator::global()->bounded(1, 3);
    if (liste.contains(i))
8   {
        if (liste.contains(i+1))
10  {
            liste.append(i+2);
```

## 5. Implementierung

```
12         }
13         else
14         {
15             liste.append(i+1);
16         }
17     }
18     else
19     {
20         liste.append(i);
21     }
22
23     if (!liste.contains(1)) {liste.append(1);}
24     else if (!liste.contains(2)) {liste.append(2);}
25     else if (!liste.contains(3)) {liste.append(3);}
26     else if (!liste.contains(4)) {liste.append(4);}
27     return liste;
```

Im zweiten Schritt wird eine Zufallszahl zwischen 1 und 3, wenn sie noch nicht enthalten ist, in die Liste geschrieben oder um eins inkrementiert und in die Liste geschrieben (Listing 5.11 Zeile 4f).

Um die nächste Zahl hinzuzufügen wird im dritten Schritt (Listing 5.11 Zeile 6-21) eine Zufallszahl zwischen 1 und 2 erzeugt und in die Liste geschrieben, sollte sie nicht vorhanden sein. Wenn sie schon existiert wird sie um eins inkrementiert und erneut überprüft ob die inkrementierte Zahl in der Liste ist. Aufgrund der Maximalgröße von 4 kann der Algorithmus so alle Fälle abdecken.

Zuletzt wird die noch fehlende Zahl der Liste ergänzt (Listing 5.11 Zeile 22ff) und die Liste zurückgegeben.

### 5.7. Mainwindow

### 5.8. Main

StateMachine Connections beschreiben / erwähnen - genau eine funktion zu einem State

### 5.9. Weitere Klassen

## 5. Implementierung

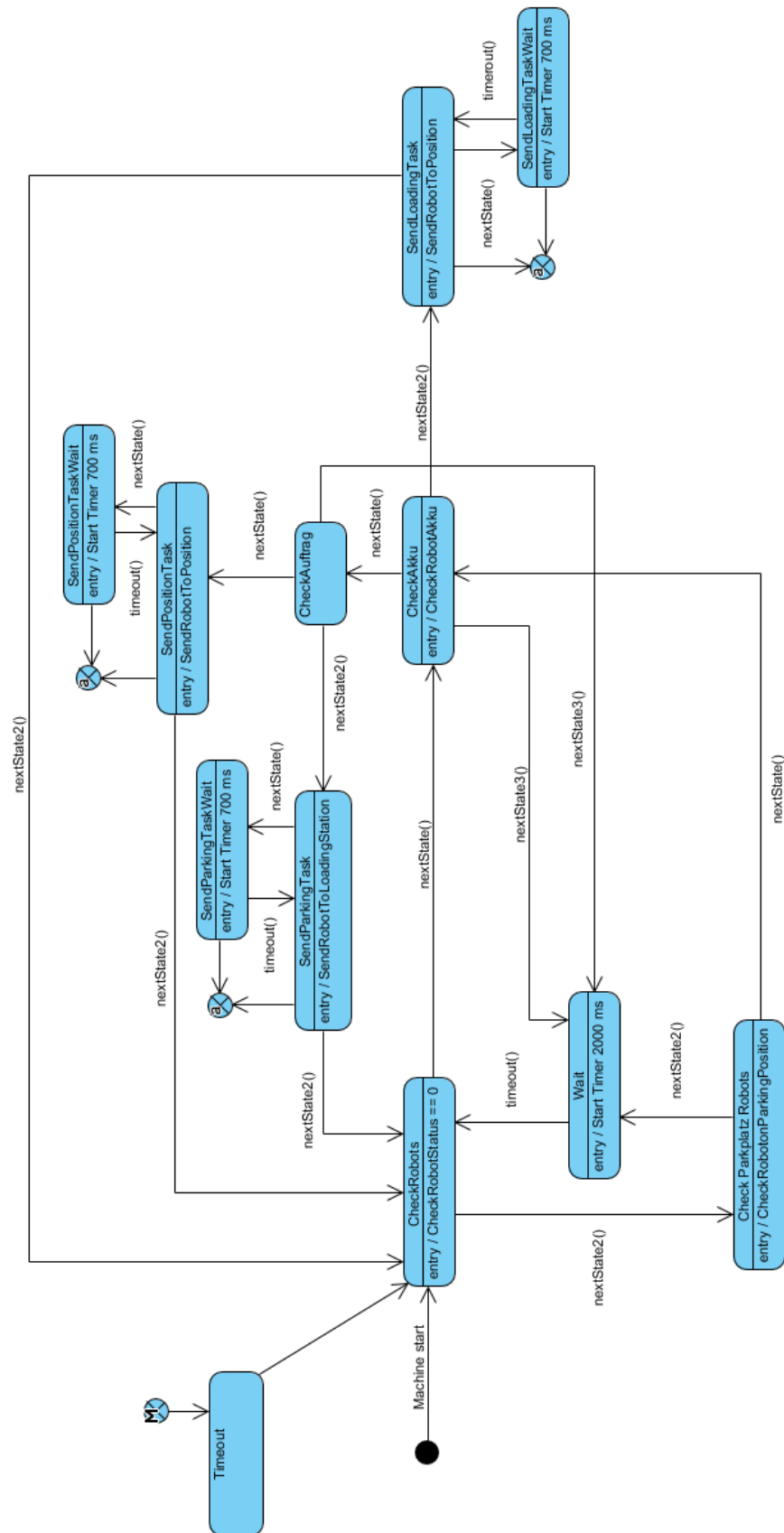


Abbildung 5.5.: Zustandsdiagramm der Auftragsplanung

## 6. Visualisierung

Die Auftragsplanungssoftware besitzt neben der Logik eine Visualisierung. Diese wurde mit den von Qt bereitgestellten Widgets grafisch programmiert und in der Funktion `mainwindow` verknüpft und organisiert.

### 6.1. Design

Die Leitlinie des Grafikdesigns der HAW war:

„Fokussiert. Direkt. Verständlich. Und vor allem sympathisch.“

„Idee der neuen visuellen Sprache ist die Reduktion auf das Wesentliche. Ziel soll es sein, einfach und klar die wichtigste Aussage zu kommunizieren.“ [Cor17, 4]

Beim Design der Software wurde versucht diese Aspekte wieder aufzugreifen. Die Visualisierung sollte ebenso direkt und verständlich sein. Vor allem die Reduktion auf das Wesentliche lag besonders im Fokus.

Um die Visualisierung im Stil der HAW zu gestalten wurde das Corporate Design bei Farb und Schriftwahl verwendet. Somit sind alle Schriftarten in der Visualisierung Open Sans. Als Farben wurden die vorgegebenen Farbstile „HAW Hauptblau“ als Umrahmung und Abtrennung aller Bereiche gewählt. Das „HAW Hellblau“ ist innerhalb der verschiedenen Bereiche als Trennung der Bedienelemente zu sehen. Da dieses blasser ist können alle Bedienelemente gut erkannt und verwendet werden.



Abbildung 6.1.: VPJ-Logo

Das entworfene VPJ-Logo (Abb. 6.1) welches in Taskmanager, Taskleiste, an oberer Bildschirmseite und zentral in der Mitte der Visualisierung zu finden ist, unterliegt ebenfalls den Richtlinien, die im Corporate Design (vgl. [Cor17]) festgehalten sind.

### 6.2. Struktur

Die Visualisierung lässt sich in insgesamt 6 Bereiche unterteilen. Die Bereiche sind in Abbildung 6.2 dargestellt. Bereich I oben links zeigt die Fertigungsstraße mitsamt Robotern, Parkplätzen und Stationen. Dieser Bereich wird folgend Live-View genannt.

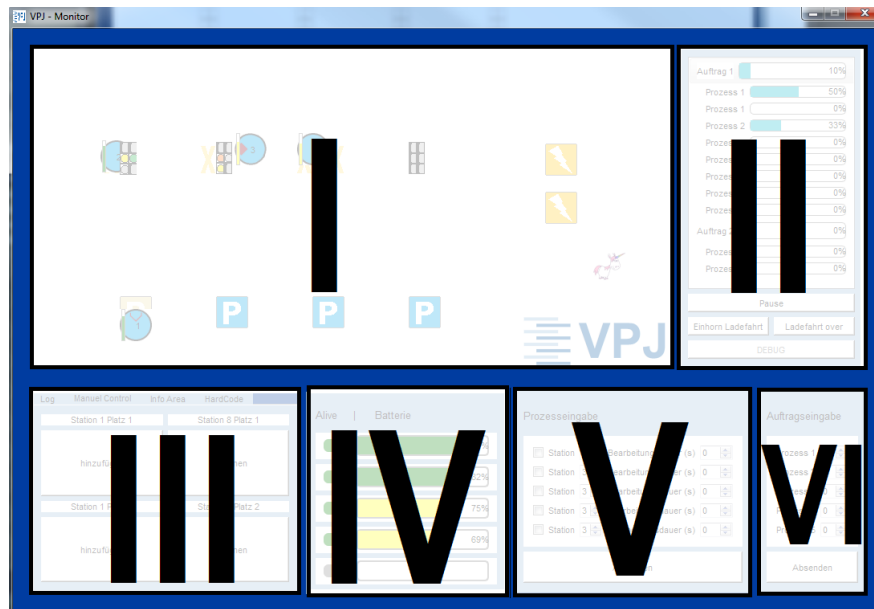


Abbildung 6.2.: Übersicht über die Bereiche innerhalb der Visualisierung

In Bereich II, der Auftragsübersicht, können die Fortschritte der laufenden Aufträge und Prozesse überwacht werden.

Innerhalb des Bereich III können über Tabs vier weitere Fenster aufgerufen werden. Dabei dienen zwei der Anzeige spezifizierter Informationen, wohingegen die anderen beiden Tabs der Kontrolle dienen. Die Kombination aller vier Tabs wird folgend Tab-View genannt.

Bereich IV zeigt den Status der Robotinos an.

Die beiden Bereiche V und VI werden ausschließlich für die Erzeugung neuer Aufträge oder Prozesse verwendet.

In Abbildung 6.3 ist eine Übersicht der Visualisierung im laufenden Betrieb dargestellt.

In den folgenden Kapiteln werden die einzelnen Bereiche der Abbildung näher beleuchtet und ihre Implementierung in der MainWindow Funktion erläutert.

### 6.3. Live-View

Der größte der Bereiche beinhaltet eine Live-View der Fertigungsstraße, dargestellt in Abbildung 6.4. Dabei sind alle Stationen, die Robotinos, Parkplätze und

## 6. Visualisierung

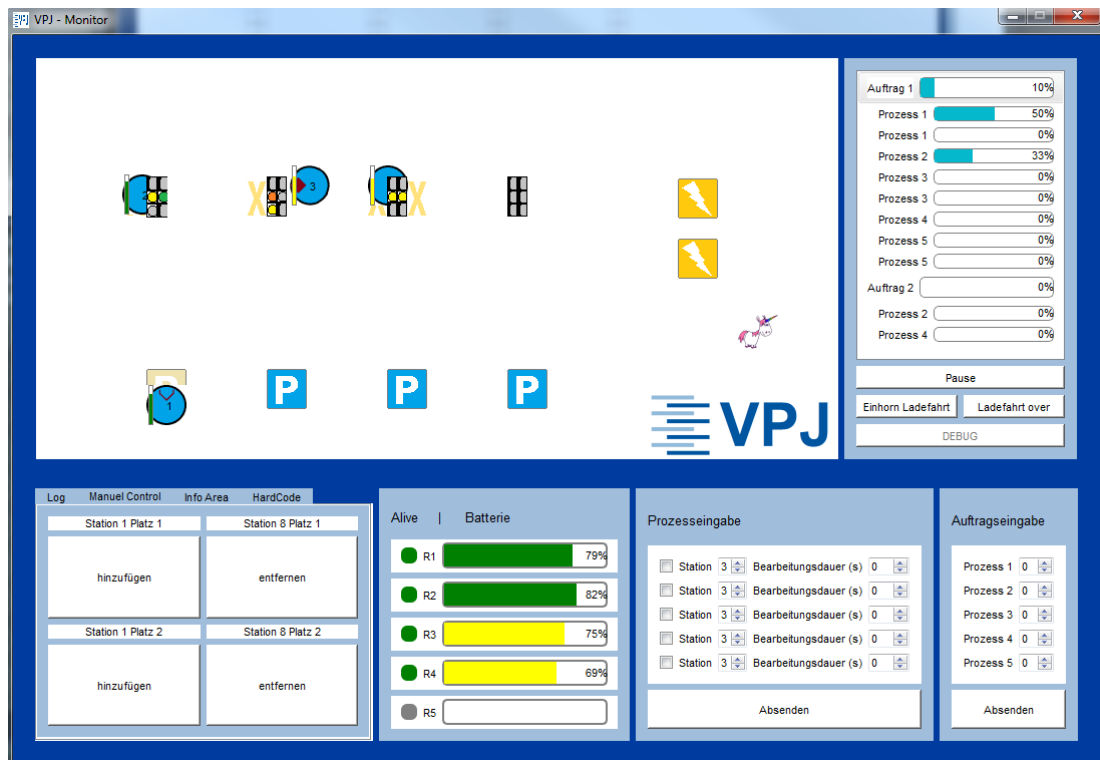


Abbildung 6.3.: Übersicht Visualisierung

die Ladestationen dargestellt. In der unteren rechten Ecke ist das VPJ-Logo eingebettet. Das Fenster hat feste Maße von 800 x 400 Pixeln, wodurch die realen Raumabmessungen einfach dargestellt werden können. 1 Pixel entspricht im realen Raum 1 cm.



Abbildung 6.4.: Visualisierung Live-View

### 6.3.1. Robotino

Der Robotino wird immer an der Position dargestellt, die in der zugehörigen Robotinoklasse hinterlegt ist. Über die Funktion `UpdateRobotinoPosition()` kann

## 6. Visualisierung

eine Aktualisierung angefordert werden. Sobald der UDP-Handler neue Positionsdaten für einen Robotino empfängt und die neue Position mindestens 2 Pixel von der Alten entfernt liegt, wird diese aktualisiert.

Die Funktion `UpdateRobotinoPosition()` prüft zunächst ob der Robotino ein Hindernis erkannt hat. Bei Vorliegen eines Hindernis wird für den Robotino ein Bild geladen, welches im Rand den Hindernistyp kodiert. In Abbildung 6.5 sind alle auftretenden Hindernistypen dargestellt. Bei einem unbekannten Hindernis wird der Robotinorand Rot dargestellt, wie rechts abgebildet. Ein noch nicht klassifiziertes Hindernis wird, wie links abgebildet, mit gelbem Rand dargestellt. In der Mitte der Abbildung ist der Robotino mit orangenem Rand abgebildet, der bei einem anderen Robotino als Hindernis angezeigt wird. Wenn kein Hindernis erkannt wird bleibt der Rand schwarz.

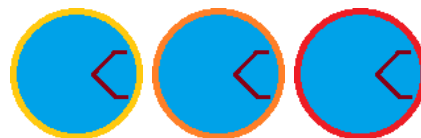


Abbildung 6.5.: Robotino Darstellung bei verschiedenen Hinderniserkennungen

Die Darstellung des Robotinos hängt neben den Hindernistypen auch von verschiedenen Stati ab. So wird ein defekter Robotino mit einer Flamme gekennzeichnet (Abb. 6.6 mittig rechts). Das Einhorn als spezieller Robotino wird ebenfalls in der Darstellung als Einhorn abgebildet, in der Abbildung rechts dargestellt.

Im Standardfall wird der Robotino wie in Abbildung 6.6 links abgebildet angezeigt. Hierbei wird unterschieden ob der Greifer des Robotinos offen (links) oder geschlossen (rechts) ist.



Abbildung 6.6.: Roboter in verschiedenen Stati (vlnr: Greifer offen, Greifer geschlossen, Defekt, Einhorn)

Nachdem das korrekte Bild für den Robotino ausgewählt wurde wird dieses gleich der Ausrichtung des Robotinos gedreht. In Listing 6.1 ist die Umsetzung im Code zu sehen. Zunächst wird eine Rotationsmatrix `rm` in Zeile 1 erzeugt. Diese Matrix wird anhand des Winkels des Robotinos konfiguriert. Anschließend wird der Bild des Robotinos in Zeile 3 gedreht und in Zeile 4 angewendet. In Zeile 6 ist die Positionierung des gedrehten Bildes abgebildet. Da die Y-Achse des Robotino invertiert zu der Darstellung ist muss diese als Offset zu 400 angegeben werden.

Listing 6.1: Robotino Drehung

```
QMatrix rm;  
2 rm.rotate(-newPosition[2]);  
QPixmap pixmap = Rlabel->pixmap()->copy().transformed(rm);  
4 Rlabel->setPixmap(pixmap);
```



## 6. Visualisierung

```
6 Robotino->move(newPosition[0]/10, 400-(newPosition[1]/10));
```

Zusätzlich zur Position beinhaltet jeder Robotino an der linken Seite einen Akkustand

UpdateRobotinoAkku UpdateRobotinoAlive UpdateRobotinoPosition UpdateRobotinoPixmap UpdateRobotinoXPixmap RobotClicked

### 6.3.2. Parkplätze und Ladestationen



Abbildung 6.7.: Visualisierung Parkplatz

ParkplatzClicked UpdateParkplatz



Abbildung 6.8.: Visualisierung Ladestation

### 6.3.3. Stationen

StationClicked UpdateStationsplatz UpdateStation

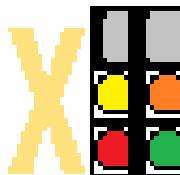


Abbildung 6.9.: Station mit verschiedenen Arbeitsplatzstati

## 6.4. Auftragsübersicht

`on_AuftragsListWidget_itemClicked`  
`on_AuftragsListWidget_itemDoubleClicked`  
`AddAuftragsI`  
`on_pushButton_clicked`  
`on_pushButton_2_clicked`  
`on_pause_clicked`  
`on_ButtonAuftragBlocked_clicked`

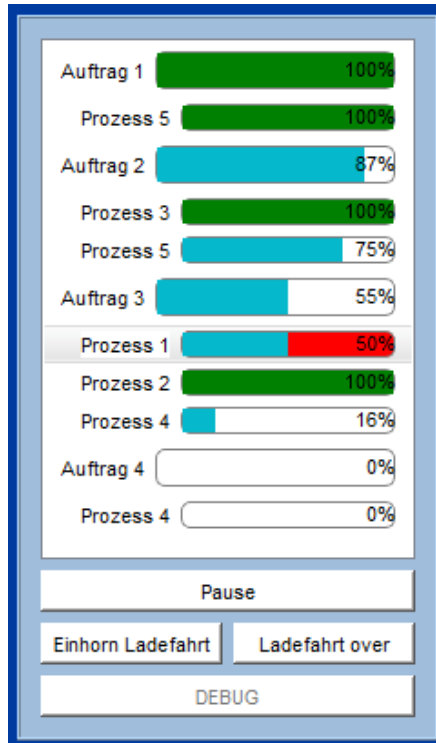


Abbildung 6.10.: Auftragsfortschritt

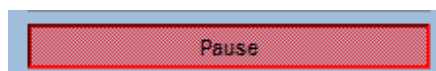


Abbildung 6.11.: Pause Button gedrückt

## 6.5. Tab-View

### 6.5.1. Log-View

### 6.5.2. Manual Control

`on_S8P2weg_clicked...`

### 6.5.3. Timestamp-Area

`SetTimestamps`

## 6. Visualisierung

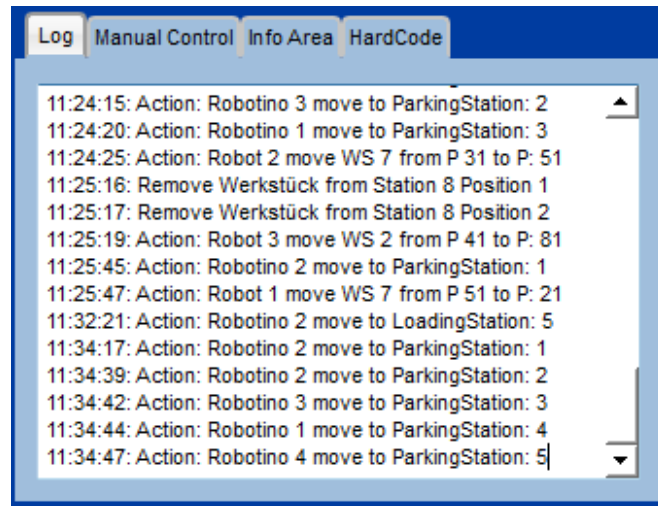


Abbildung 6.12.: Tab: Log View

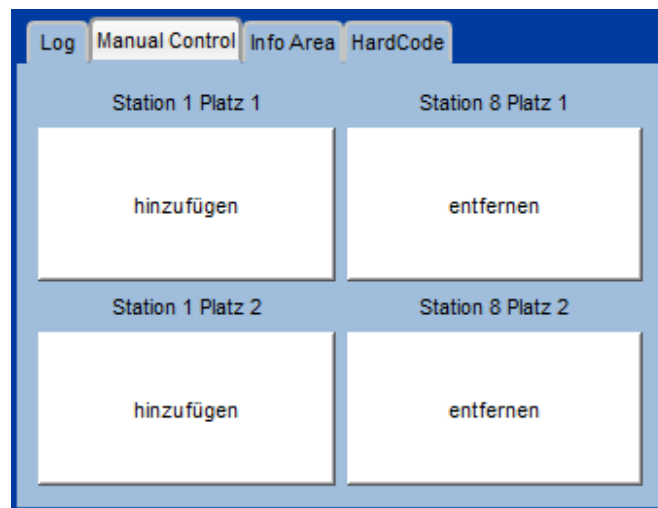


Abbildung 6.13.: Tab: Manual Control

### 6.5.4. Hard-Code Bereich

GetPressedHardCodeButton ResetButtonsExeptOne on\_ButtonDefektPlatz\_clicked...RobotClick

## 6.6. Roboterstatus

UpdateRobotinoAlive UpdateRobotinoAkku ToggleAliveRobotino

## 6.7. Prozesseingabe

on\_endProzess\_clicked

## 6. Visualisierung

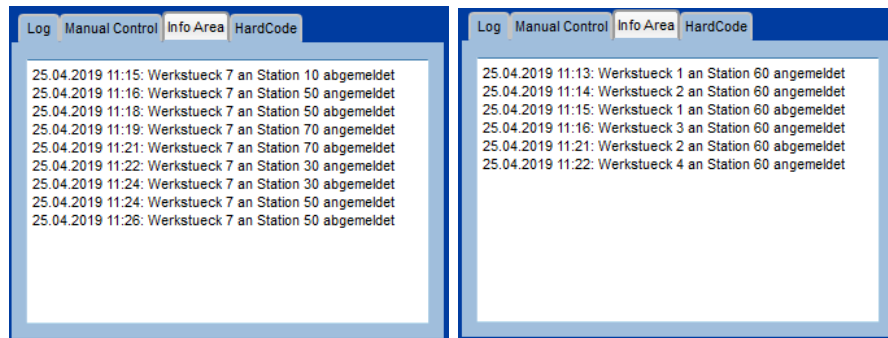


Abbildung 6.14.: Tab: Info Area

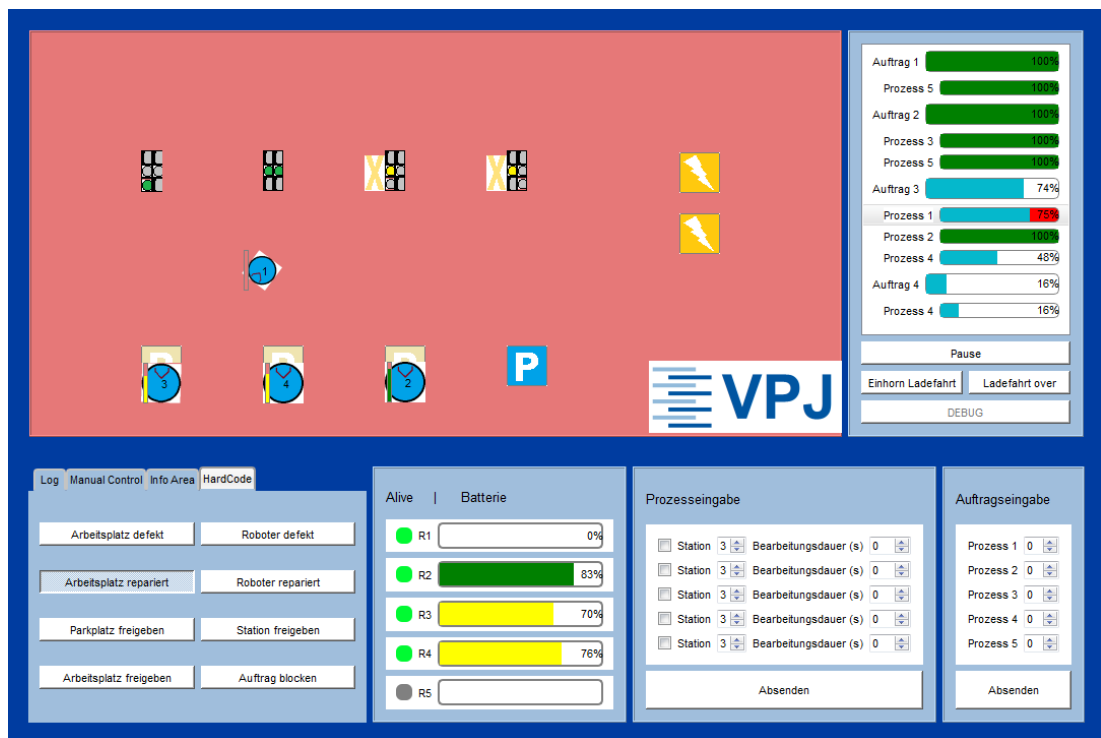


Abbildung 6.16.: Übersicht Visualisierung im Hard-Code Modus

## 6.8. Auftragseingabe

AddAuftragsItem on<sub>endAuftrag</sub>clicked

## 6.9. Benutzerinteraktion

### 6.9.1. Tooltips

UpdateStationToolTip

Tooltips mit Bildern her und erkläeren

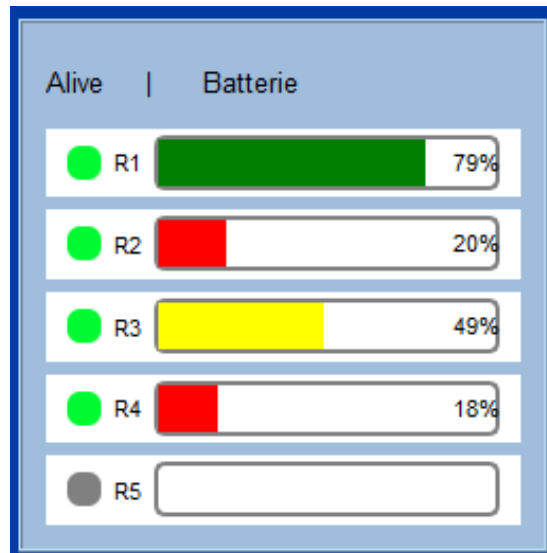


Abbildung 6.17.: Batterie und Statusanzeige

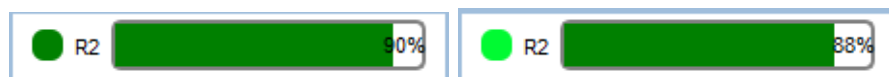


Abbildung 6.18.: Roboterstatus-LED blinkend

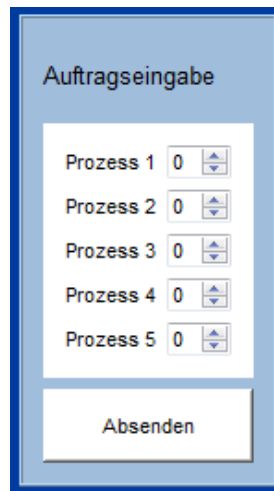
## 6.10. MainWindow

Neben allen bisher bereits beschriebenen Funktionen wird folgend eine Übersicht über die MainWindow-Funktion gegeben.

Station	Bearbeitungsdauer (s)
<input checked="" type="checkbox"/> Station 3	20
<input checked="" type="checkbox"/> Station 4	0
<input type="checkbox"/> Station 3	10
<input type="checkbox"/> Station 5	5
<input type="checkbox"/> Station 6	0

Absenden

Abbildung 6.19.: Visualisierung - Prozesseingabe



The image shows a web form titled "Auftragseingabe" (Order Entry). It contains five input fields, each labeled "Prozess 1" through "Prozess 5". Each field has a text input box showing the value "0" and a small spinner control to its right. Below these fields is a large button labeled "Absenden" (Submit).

Prozess	Wert
Prozess 1	0
Prozess 2	0
Prozess 3	0
Prozess 4	0
Prozess 5	0

Absenden

Abbildung 6.20.: Visualisierung - Auftragsvergabe

## 7. Simulation

Um verschiedene Programmabläufe und Funktionen zu testen war es wichtig, auch ohne realen Roboter die standardmäßige Programmfunktionalität darstellen zu können. Die Klasse `SimulatedRobot` erfüllt genau diese Anforderung.

Mittels einer `simulated`-Flag kann in der Main-Funktion ein Simulationsbetrieb gestartet werden. In diesem ist kein realer Roboter notwendig, und trotzdem findet eine normale Auftragsplanung- und Abarbeitung statt. Die `simulated`-Flag wird in der Initialisierung dem UDP-Handler übergeben, der die Kommunikation mit den Robotern übernimmt. Somit wird in der Simulation kein Socket erzeugt und verbunden (siehe 5.3), sondern die Klasse `SimulatedRobot` genutzt.

`SimulatedRobot` enthält eine State-Machine mit 13 Zuständen, die den Roboter ausreichend nachbilden. Weiterhin werden Funktionen für das simulierte Senden des Roboters an den UDPHandler und Timer für die Zustandswechsel bereitgestellt.

Wie im echten Roboter wird der aktuelle Auftrag und Auftragstyp zwischengespeichert.

### 7.1. Zustandsdiagramm simulierter Roboter

Das Zustandsdiagramm in Abbildung 7.1 beschreibt die Zustände und Transitionen eines simulierten Roboters. Das Diagramm wurde anschließend als State Machine (vgl. 2.1.4) implementiert.

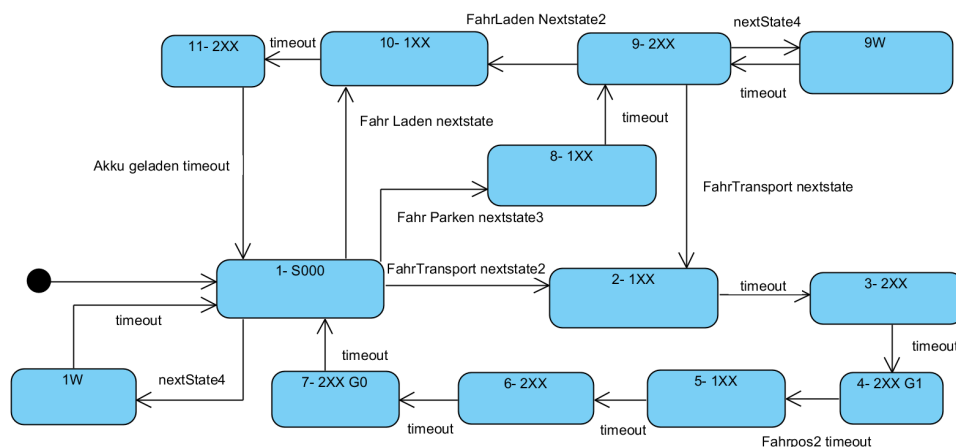


Abbildung 7.1.: Ablaufdiagramm des simulierten Roboters

## 7. Simulation

Da in der Fertigungsplanung für die Vergabe der Aufträge die tatsächliche Position des Roboters nicht berücksichtigt wird, wird im simulierten Roboter ausschließlich die für die Planung benötigte Status- und Greiferinformation nachgepflegt.

Die Änderung der Statusinformation ist im Diagramm in den Zuständen als Zahl mit zwei folgenden Dont-Care (XX) gekennzeichnet, da die Position keine weitere Relevanz hat. Eine Änderung des Greifers ist mit G0 für Greifer öffnet sich oder G1 für Greifer schließt sich gekennzeichnet.

Zur Initialisierung wurden ähnlich der State Machine aus Kapitel 5.6.5 zunächst alle Zustände und Transitionen hinzugefügt, verknüpft und die State Machine gestartet.

Der Initialzustand 1 kann über Empfang eines Auftrags zum Laden zu Zustand 10, zum Parken zu Zustand 8 oder zum Transport zu Zustand 2 verlassen werden. Außerdem wird das aktive Abfragen eines Auftrags an den Roboter alle 3 Sekunden über den Zustand 1W getriggert. Über den Wartezustand wird ein bearbeiteter noch anliegender Auftrag zurückgesetzt.

Der Ablauf innerhalb der State-Machine zwischen zwei Aufträgen erfolgt über fest definierter Timer, die Zustandswechsel bewirken und ein Senden des Roboterstatus an den UDP-Handler hervorrufen.

Da innerhalb von QT keine Events verloren gehen kann auf ein periodisches Senden, wie der echte Roboter es tut verzichtet werden.

### **Simulierte Ladefahrt**

Wenn in Zustand 1 am simulierten Roboter ein Ladefahrtauftrag anliegt, also ein Auftrag mit der ID 3, so wird direkt in Zustand 10 gewechselt. Es wird an den UDP-Handler ein Status 100 zurückgegeben (auf dem Weg zur Ladestation) und ein Timer von 3 Sekunden gestartet. Nach Ablauf des Timers wird in Zustand 11 der Status 2XX zurückgegeben, was bedeutet, dass die Ladestation erreicht wurde. Nach Ablauf weiterer 5 Sekunden gilt das Laden als beendet und der Roboter geht in Zustand 1 über und sendet Status 000.

### **Simulierter Parkvorgang**

Bei einem Parkauftrag in Zustand 1, also ein Auftrag mit der ID 2, wird in Zustand 8 gewechselt. An den UDP-Handler wird, da der simulierte Roboter jetzt auf dem Weg zum Parken ist, als Status 100 zurückgegeben. Nach einer Fahrzeit von 3 Sekunden hat der Roboter sein Ziel erreicht und gibt in Zustand 9 als Status 200 an den UDP-Handler.

Zustand 9 kann entweder über einen Auftrag mit der ID 3 in Zustand 10 verlassen werden, und eine Ladefahrt wird simuliert, oder mit einem Auftrag und der ID 1 einen Transportauftrag in Zustand 2 verlassen werden.



## 7. Simulation

Über den Wartezustand 9W kann auf weitere Aufträge reagiert werden. Dieser wird alle 3 Sekunden aufgerufen.

### **Simulierter Transport**

Sobald ein simulierter Roboter in Zustand 2 kommt wird an den UDP-Handler der Status 100 übermittelt, da der Roboter sich auf dem Weg zur ersten Auftragsposition befindet. Nach 3 Sekunden wird in Zustand 3 der Status 2XX gesendet, da der Roboter an der ersten Position angekommen ist. Eine weitere Sekunde später wird das Werkstück gegriffen, was durch Senden des Greiferstatus 1 in Zustand 4 übermittelt wird. In Zustand 5 wird über Ablauf eines zweisekündigen Timers der Status 100 gesendet, da sich der Roboter auf dem Weg zur zweiten Station des Auftrags befindet.

In Zustand 6, der nach 3 Sekunden bearbeitet wird, wird zunächst der Status auf 2XX gesetzt, da der Roboter angekommen ist. Nach einer Sekunde wird über Zustand 7 der Greifer geöffnet und der Greiferstatus auf 0 zurückgesetzt. Zwei Sekunden später wird wieder in Zustand 1 auf einen neuen Auftrag gewartet.

## 8. Datenbank

# Abbildungsverzeichnis

1.1. CMUcam5 Pixy	
Quelle: <a href="http://charmedlabs.com/default/products/">http://charmedlabs.com/default/products/</a> . . . . .	1
1.2. PixyCam Gehäusefehler . . . . .	2
4.1. Sequenzdiagramm . . . . .	6
4.2. Positionskodierung . . . . .	7
5.1. Klassendiagramm . . . . .	10
5.2. Visualisierung des Rückgabewertes der Funktionen getNextUnfinishedProzessstep() und GetActualProzessstep() . . . . .	19
5.3. Diagramm Fahrweganalyse . . . . .	22
5.4. Diagramm Stationsanalyse . . . . .	23
5.5. Zustandsdiagramm der Auftragsplanung . . . . .	31
6.1. VPJ-Logo . . . . .	32
6.2. Übersicht über die Bereiche innerhalb der Visualisierung . . . . .	33
6.3. Übersicht Visualisierung . . . . .	34
6.4. Visualisierung Live-View . . . . .	34
6.5. Robotino Darstellung bei verschiedenen Hinderniserkennungen . . . . .	35
6.6. Roboter in verschiedenen Stati (vlnr: Greifer offen, Greifer geschlossen, Defekt, Einhorn) . . . . .	35
6.7. Visualisierung Parkplatz . . . . .	36
6.8. Visualisierung Ladestation . . . . .	36
6.9. Station mit verschiedenen Arbeitsplatzstati . . . . .	36
6.10. Auftragsfortschritt . . . . .	37
6.11. Pause Button gedrückt . . . . .	37
6.12. Tab: Log View . . . . .	38
6.13. Tab: Manual Control . . . . .	38
6.14. Tab: Info Area . . . . .	39
6.16. Übersicht Visualisierung im Hard-Code Modus . . . . .	39
6.17. Batterie und Statusanzeige . . . . .	40
6.18. Roboterstatus-LED blinkend . . . . .	40
6.19. Visualisierung - Prozesseingabe . . . . .	40
6.20. Visualisierung - Auftragsvergabe . . . . .	41
7.1. Ablaufdiagramm des simulierten Roboters . . . . .	42

# Literaturverzeichnis

- [Cor17] Hochschule für angewandte Wissenschaften Hamburg. *Corporate Design Manual*, 2017. Stand am 09.05.2019:  
[https://www.haw-hamburg.de/fileadmin/user\\_upload/Presse\\_und\\_Kommunikation/Downloads/Corporate\\_Design\\_Manual\\_HAW\\_Hamburg-2017-05-10.pdf](https://www.haw-hamburg.de/fileadmin/user_upload/Presse_und_Kommunikation/Downloads/Corporate_Design_Manual_HAW_Hamburg-2017-05-10.pdf).
- [Qt 19a] The Qt Company Ltd. *QSqlDatabase Class*, 2019. Stand am 09.05.2019: <https://doc.qt.io/qt-5/qsqldatabase.html>.
- [Qt 19b] The Qt Company Ltd. *QUdpSocket Class*, 2019. Stand am 09.05.2019: <https://doc.qt.io/qt-5/qudpsocket.html>.
- [Qt 19c] The Qt Company Ltd. *The State Machine Framework*, 2019. Stand am 09.05.2019:  
<https://doc.qt.io/qt-5/statemachine-api.html>.
- [Sch09] Schober. *Festo Dokumentation Handbuch MPS Transfersystem*. Festo Didactic, 2009.

# A. Inhalt der CD

- Dieses Dokument als PDF „VPJ.pdf“
- Alle konstruierten Bauteile sowohl als STL-, als auch als CATIA-Part-Datei im Ordner „Konstruktion“
- Den Quellcode aller Anwendungen im Ordner „Code“
  1. config.ini
  2. config\_bandposition.py
  3. config\_farbton.py
  4. config\_referenzposition.py
  5. config\_playground.py - als Template und zum Testen von Einstellungen
- Alle Abbildungen der Arbeit im Ordner „Abbildungen“
- Ein Video „LED\_Video.mp4“ zur Veranschaulichung der LEDs des Raspberry Pi
- Ein Video „Raspberry Pi Test.mp4“ zur Veranschaulichung der Anwendung des Raspberry Pi auf dem Festo-Transfersystem
- Ein Video „Pixy Test auf Anlage.mp4“ zur Veranschaulichung der Anwendung der PixyCam auf dem Festo-Transfersystem
- Marktrecherche ImageProcessingPlatform.pdf  
„How to choose the best embedded processing platform for on-board UAV image processing ?“ von Dries Hulens, Jon Verbeke und Toon Goedem