

# Dokumentation

Daniel Friedrich  
Jan-Henrik Meyer

Verbundprojekt 2018/2019

Daniel Friedrich  
Jan-Henrik Meyer

## Verbundprojekt 2018/2019

Abschlussdokumentation 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

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

Eingereicht am: 13. Juni 2019

**Daniel Friedrich    Jan-Henrik Meyer**

**Thema der Arbeit**

Verbundprojekt 2018/2019

**Stichworte**

Qt, Datenbank, MYSQL, Auftragsplanung, Visualisierung

**Kurzzusammenfassung**

In dieser Arbeit wurde ein Protokoll zur Versendung von Aufträgen entworfen. In einem entwickelten Programm in der neuen Umgebung Qt wurde eine Planung und Verwaltung aller Aufträge, sowie eine Visualisierung der Produktionsanlage umgesetzt. Es wurden Schnittstellen entworfen und implementiert, die den Fertigungsrechner, den Navigationsrechner, den Fertigungsplanungsrechner verbinden. Eine Datenbank wurde entworfen und implementiert. Typische Fehler in der Produktion wurden abgefangen oder vorab vermieden.

**Daniel Friedrich    Jan-Henrik Meyer**

**Title of Thesis**

Verbundprojekt 2018/2019

**Keywords**

Qt, database, MYSQL, job planning, visualization

**Abstract**

In this thesis, a protocol for the transmit of orders was designed. In a developed program in the new environment Qt was implemented a planning and administration of all orders, as well as a visualization of the production system. Interfaces were designed and implemented, that connect the production computer, the navigation computer and the production planning computer. A database has been designed and implemented. Typical errors in production were intercepeted or avoided in advance.

# Glossar

Begriff	Beschreibung
Auftrag	Ein Auftrag ist das vom Benutzer eingegebene Konstrukt aus verschiedenen Prozessen
Arbeitsplatz	Ein einzelner Bearbeitungsplatz an einer Station
Einhorn	Eigenname von Robotino 5
Fertigungsstraße	Zusammenfassung aller Stationen und Arbeitsplätze
Fertigungsrechner	Rechner mit Datenbank und Kommunikation zu RFID
Fertigungsplanungsrechner	Rechner mit UDP-Kommunikation und Auftragsplanung
Position	Enthält eine Kodierung einer Station oder Arbeitsplatzes
Produktionsprozess	Ein Produktionsprozess ist im Zusammenhang mit der Datenbank als Rezept für die Bearbeitung des Werkstücks zu verstehen.
Prozess	Ein Prozess beschreibt die Abfolge unterschiedlich vieler Prozessschritte. Im Zusammenhang der Datenbank wird der Begriff Werkstück verwendet
Prozessschritt	Enthält genau einen Arbeitsplatz mit Bearbeitungsdauer
Station	Der Zusammenschluss aus einem RFID Lese- Schreibgerät und den dahinterliegenden beiden Arbeitsplätzen
Werkstück	Ein Werkstück symbolisiert exakt einen „Stein“ in der Fertigungsstraße. Ein Werkstück entspricht immer genau einem Prozess

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Gewerk 1: RFID, Fertigungsrechner, Fertigungsplanung-/Steuerung	1
1.2. Gesamtkonzept Gewerk 1 . . . . .	2
<b>2. Grundlagen und Konzepte von Qt</b>	<b>3</b>
2.1. Eventbasiertes System . . . . .	3
2.2. Sockets . . . . .	4
2.3. Datenbank in Qt . . . . .	5
2.4. State Machines in Qt . . . . .	5
2.5. QListWidget . . . . .	7
<b>3. Schnittstelle zum Robotino</b>	<b>8</b>
3.1. Sequenzdiagramm . . . . .	8
3.2. Telegramme . . . . .	9
3.2.1. Telegramm vom Fertigungsplanungsrechner zu Gewerk 2 . . . . .	9
3.2.2. Positions kodierung . . . . .	10
3.2.3. Telegramm vom Robotino zum Fertigungsplanungsrechner . . . . .	11
3.3. Fehlertypen und Behebung . . . . .	12
<b>4. Implementierung</b>	<b>14</b>
4.1. Programmstruktur . . . . .	14
4.2. Database-Handler . . . . .	15
4.2.1. Konstruktor, Initialisierung und Sendefunktion . . . . .	15
4.2.2. Getter . . . . .	17
4.2.3. Updates . . . . .	17
4.2.4. Setter . . . . .	18
4.3. UDP-Handler . . . . .	18
4.3.1. UDP-Handler für Navigationsrechner . . . . .	18
4.3.2. UDP-Handler für Robotino . . . . .	20
4.4. Robotino . . . . .	22
4.4.1. Werte aktualisieren . . . . .	22
4.4.2. Visualisierungsfunktionen . . . . .	22
4.5. Auftrag, Prozess, Prozessschritt . . . . .	23
4.5.1. Prozessschritt . . . . .	23
4.5.2. Prozess . . . . .	24
4.5.3. Auftrag . . . . .	25
4.6. Fertigungsplanung . . . . .	26
4.6.1. Initialisierung . . . . .	26
4.6.2. Roboter Statusänderungen . . . . .	29
4.6.3. Hard-Code Funktionen . . . . .	30
4.6.4. Zustandsdiagramm . . . . .	31

## Inhaltsverzeichnis

4.6.5. State Machine Implementierung . . . . .	31
4.6.6. Shuffle-Algorithmus . . . . .	35
4.7. Weitere Klassen . . . . .	37
4.7.1. Main . . . . .	37
4.7.2. Timestamp . . . . .	39
4.7.3. Fertigungsstrasse . . . . .	39
4.7.4. Auftrags- und Prozessitem . . . . .	40
<b>5. Visualisierung</b>	<b>41</b>
5.1. Design . . . . .	41
5.2. Struktur . . . . .	42
5.3. Live-View . . . . .	42
5.3.1. Robotino . . . . .	43
5.3.2. Parkplätze und Ladestationen . . . . .	45
5.3.3. Stationen . . . . .	46
5.4. Auftragsübersicht . . . . .	46
5.4.1. Auftragsdarstellung . . . . .	47
5.4.2. Buttons der Auftragsübersicht . . . . .	50
5.5. Tab-View . . . . .	50
5.5.1. Log-View . . . . .	50
5.5.2. Manual Control . . . . .	53
5.5.3. Timestamp-Area . . . . .	53
5.5.4. Hard-Code Bereich . . . . .	54
5.6. Roboterstatus . . . . .	56
5.6.1. Akkustand . . . . .	57
5.6.2. Alive-Status . . . . .	58
5.7. Prozesseingabe . . . . .	58
5.8. Auftragseingabe . . . . .	59
5.9. Tooltips . . . . .	61
5.9.1. Werkstücktooltip . . . . .	61
5.9.2. Prozessitemtooltip . . . . .	62
<b>6. Robotersimulation</b>	<b>63</b>
6.1. Zustandsdiagramm simulierter Roboter . . . . .	63
<b>7. Fertigungsrechner</b>	<b>66</b>
<b>8. Datenbank</b>	<b>67</b>
8.1. Aufgabe der Datenbank . . . . .	67
8.2. Konzept . . . . .	67
8.3. Konzeptionelles Modell . . . . .	67
8.3.1. Normalisierung . . . . .	70
8.4. Relationales Datenbankmodell . . . . .	71
8.5. Zugriff auf die Datenbank . . . . .	71
8.6. Umsetzung des entwickelten Modells . . . . .	72
8.7. MySQL-Befehle für die Fertigungsplanung . . . . .	75
8.7.1. Getter . . . . .	75
8.7.2. Update . . . . .	76
8.7.3. Setter . . . . .	76

## *Inhaltsverzeichnis*

<b>9. Fertigungsüberwachung</b>	<b>78</b>
9.1. Konzept . . . . .	78
9.2. Programmstruktur . . . . .	78
9.3. Schnittstelle der RFID-Schreib-Lese-Köpfen . . . . .	80
9.4. Programmteile . . . . .	80
9.4.1. Main PLC_PRG . . . . .	81
9.4.2. FB Aufruf_RFID_Channel_2_bis_8_Lesen . . . . .	81
9.4.3. FB Aufruf_RFID_Channel_1_Schreiben . . . . .	82
9.4.4. FB FB_RFID_SS15 . . . . .	82
9.4.5. FB Werkstueck_Tagen . . . . .	83
9.4.6. FB Timestamp_anlegen . . . . .	83
9.4.7. FB Datenbank_Read_Write . . . . .	85
9.4.8. FB SQL4Automation . . . . .	85
9.4.9. Einstellen der Systemzeit . . . . .	87
<b>10. Zusammenfassung</b>	<b>89</b>
<b>A. Inhalt der CD</b>	<b>93</b>
<b>Selbstständigkeitserklärung</b>	<b>94</b>

# 1. Einleitung

Im Verbundprojekt Autonome Systeme soll ein Fertigungsprozess mit autonom agierenden Robotern automatisiert werden. Dazu befinden sich acht Stationen mit jeweils zwei Arbeitsplätzen und einem Platz zum Lesen oder Schreiben von RFID-Tags im Versuchsaufbau. Station 1 repräsentiert dabei das Rohteil lager und Station 8 das Fertigteil lager. Die Arbeitsschritte werden durch Aufenthalt der Werkstücke für eine bestimmte Zeit an den Arbeitsplätzen simuliert. Je nach Produkt, das hergestellt werden soll, variiert die Anzahl, Reihenfolge, und Bearbeitungszeit der Stationen. Zum Transport der Werkstücke zwischen den Stationen werden autonom agierende Roboter mit einer Greifzange eingesetzt. Die Navigation der Roboter wird mit Hilfe von ermittelten Positionsdaten realisiert, die mit Kameras ermittelt werden. Als Software wird, für die Programmierung der Roboter, Matlab/Simulink eingesetzt. Für die Auftragsplanung wird Qt verwendet. Das Speichern der Produktionsdaten erfolgt über eine MySQL-Datenbank und die RFID-Technologie wird mit CODESYS angesteuert.

## 1.1. Aufgaben Gewerk 1

Das Gewerk 1 ist zuständig für:

- die Fertigungsplanung,
- die Visualisierung des Fertigungsprozesses,
- das Beschreiben und Auslesen der RFID-Tags auf den Werkstücken und
- das Ablegen aller produktionsrelevanten Daten in der MySQL-Datenbank.

Die Fertigungsplanung soll die über das visuelle HMI eingegebenen Aufträge verwalten und den Robotern ihre Aufträge zuweisen. Die Visualisierung soll die Roboter und ihre Position so wie die Arbeitsstationen darstellen. Über die Visualisierung soll ebenfalls die Auftragseingabe stattfinden und der aktuelle Fortschritt eines Auftrags abgebildet werden. Ebenso ist die vom Fertigungsrechner mit Hilfe der RFID-Technologie durchgeführte Verfolgung der Werkstücke darzustellen.

Der Fertigungsrechner soll die RFID-Tags der Werkstücke mit Hilfe der RFID-Schreib-Lese-Köpfe beschreiben, beziehungsweise auslesen, um so eine Verfolgung der Werkstücke durch den Produktionsprozess zu ermöglichen. Auf dem Fertigungsrechner befindet sich auch der Server für die MySQL-Datenbank, die alle produktionsrelevanten Daten speichert und sowohl vom Programm für die RFID-Technologie, als auch von der Fertigungsplanung lesend und schreibend angesprochen wird.

## 1. Einleitung

### 1.2. Gesamtkonzept Gewerk 1

Das Gesamtkonzept von Gewerk 1 ist in Abbildung 1.1 dargestellt.

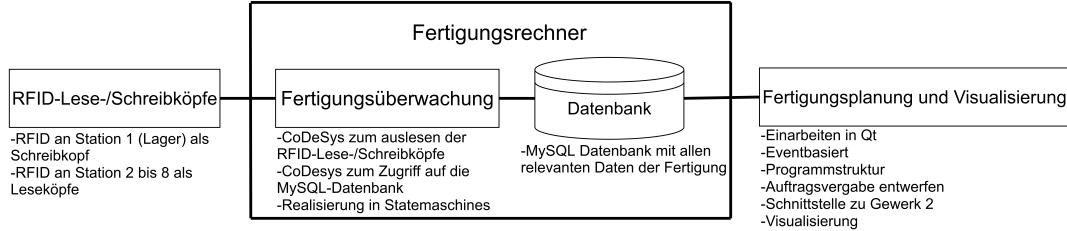


Abbildung 1.1.: Gesamtkonzept von Gewerk 1

Die Fertigungsplanung und die Visualisierung werden in einem gemeinsamen Programm, dass mit Qt geschrieben wurde, realisiert. Die Fertigungsplanung empfängt für die Visualisierung über Ethernet per UDP/IP die Daten des Navigationsrechners. Die Kommunikation mit den Robotern wird ebenso von der Fertigungsplanung über WLAN per UDP/IP realisiert. Die Kommunikation zur Datenbank, die sich auf dem Fertigungsrechner befindet, findet über Ethernet statt. Die RFID-Schreib-Lese-Köpfe werden über ein Interface mittels Profibus-DP an den Fertigungsrechner angeschlossen. Das Programm zum Auslesen, beziehungsweise Beschreiben, der RFID-Tags wird mit CODESYS geschrieben und läuft auf einer Soft-SPS auf dem Fertigungsrechner. An Station 1 werden die RFID-Tags des Werkstücks beschrieben und an den Stationen 2 bis 8 ausgelesen. Über SQL4Automation kann das Programm auf die Datenbank lesend und schreibend zugreifen. In der Datenbank sind alle für den Produktionsprozess relevanten Daten abgelegt. Es handelt sich um eine relationale MySQL-Datenbank. Die Datenbank dient auch zur Kommunikation zwischen dem CODESYS-Programm auf dem Fertigungsrechner und der Fertigungsplanung.

# 2. Grundlagen und Konzepte von Qt

In diesem Kapitel werden einige Grundlagen erläutert, die zum Verständnis der späteren Kapitel der Arbeit beitragen.

Qt ist eine plattformübergreifende Anwendungsentwicklungsumgebung für Desktop-anwendungen, eingebettete und mobile Systeme. Unterstützt werden unter anderem Linux, OS X, Windows, Android, iOS, BlackBerry oder Sailfish OS.

Die Umgebung ist in C++ geschrieben, weshalb vorwiegend in dieser Sprache programmiert wird. Eine Besonderheit an Qt ist die Benutzung eines Signal-Slot-Systems.

Ähnlich anderen Entwicklungsumgebungen bietet Qt mit einem implementierten Designer, Möglichkeiten zur GUI-Programmierung mit vorgefertigten Widgets (vgl. [Qt 19a]).

Die nächsten Abschnitte zeigen einige Besonderheiten von Qt und Konzepte, die in der Ausarbeitung des Projekts verwendet wurden.

## 2.1. Eventbasiertes System

Innerhalb von Qt werden Signale und Slots zur Kommunikation zwischen Objekten genutzt. Das Signal-Slot Konzept ersetzt das in anderen Umgebungen verwendete Callback-System, welches beispielsweise in Eclipse bei der Java-Programmierung oder in Visual Studio mit C# verwendet wird.

In Abbildung 2.1 ist das Konzept visuell dargestellt. Ein Signal wird immer dann emittiert, wenn ein bestimmtes Event auftritt. Die in Qt verwendeten Widgets stellen bereits viele dieser Signale bereit, können jedoch um Eigene ergänzt werden.

Das Gegenstück zu den Signalen sind Slots, welche als Antwort auf ein Signal aufgerufen werden. Es können beliebig viele Signale an einen Slot gebunden werden. Es ist ebenfalls möglich Signale direkt an weitere Signale zu binden.

Ein Slot kann wie eine normale Funktion auch mittels Funktionsaufruf verwendet werden.

Nähere Informationen zu diesem Konzept, zu Signalen und zu Slots können der Quelle [Qt 19e] entnommen werden.

## 2. Grundlagen und Konzepte von Qt

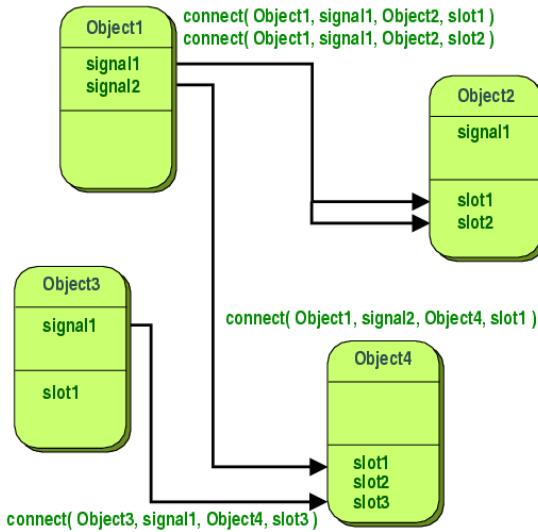


Abbildung 2.1.: Signale und Slots in Qt

Um Signale und Slots zu verbinden werden verschiedene Möglichkeiten bereitgestellt. In dieser Arbeit wurde hauptsächlich mit der `connect()` Funktion gearbeitet, die von jeder Klasse, die von `QObject` erbt, bereitgestellt wird.

Listing 2.1: Connect Signals and Slots

```
QObject ::connect( const QObject *sender , const char *signal  
, const QObject *receiver , const char *method , Qt::  
ConnectionType type = Qt::AutoConnection )
```

In Listing 2.1 ist der Connect-Aufruf vollständig dargestellt. Hinzuzufügen ist, dass der letzte Übergabeparameter, `ConnectionType`, optional ist, da er als Standardwert `QT::AutoConnection` beinhaltet. Die ersten beiden Parameter enthalten also immer die Senderseite, die letzten beiden Parameter die Empfängerseite. Dabei ist zuerst das Objekt anzugeben, welches sendet oder empfängt und danach das Signal, bzw. der Slot, der genutzt werden soll.

## 2.2. Sockets

Die Kommunikation über UDP wird in Qt mit einem `QUdpSocket` bereitgestellt. Der Socket bietet Funktionen, mit denen UDP-Datagramme gesendet und empfangen werden können.

Um einen Datenaustausch zu ermöglichen wird der Socket mittels einer `Bind()`-Methode an eine spezielle Adresse und einen Port gebunden. Vorteil dieser Implementierung ist, dass beim Erhalt einer Nachricht ein `readyRead()`-Signal emittiert wird, auf das sich verbunden werden kann.

Sobald das `readyRead()`-Signal emittiert wurde, liefert `hasPendingDatagrams()` `true` zurück. Das Datagramm kann anschließend über `readDatagram()` oder `receiveDatagram()` ausgelesen werden (vgl. [Qt 19d]).

## 2. Grundlagen und Konzepte von Qt

Außerdem unterstützt die Klasse UDP multicast.

Wenn ein Datagramm nicht ausgelesen wird, wird kein weiteres readyRead()-Signal emittiert, bis dieses ausgelesen wurde.

## 2.3. Datenbank in Qt

Um möglichst einfach Datenbanken in Qt zu implementieren, wird die Klasse QSqlDatabase (vgl. [Qt 19c]) bereitgestellt. Diese Klasse ist für die Verbindung und Kommunikation an eine bestehende Datenbank zuständig. Der Datenbanktyp wird über den eingesetzten Treiber bestimmt. In dieser Arbeit sollte auf eine Datenbank vom Typ MYSQL zugegriffen werden.

Um den Treiber der Entwicklungsumgebung von Qt bekannt zu machen war es notwendig eine MYSQL Library einzubinden. Dies konnte gelöst werden, indem die Datei „MySQLLib.dll“, welche zum Beispiel im MySQL Connector zu finden ist (Pfad: C:\Program Files\MySQL\MySQL Connector.C 6.1\lib), an einen Ort kopiert wird, dessen Pfad von der Entwicklungsumgebung erkannt wird. Ein Möglicher Ort ist C:\Windows.

Nachdem sich mit dem Setzen von Nutzernamen, Passwort, Datenbankname und Host auf die Datenbank verbunden wurde, können die gegebenen Funktionen verwendet werden.

In dieser Arbeit wurde eine QSqlQuery innerhalb der QSqlDatabase verwendet, um die Daten der Datenbank auszulesen und zu manipulieren. Die Query stellt Funktionen zur Verfügung, mit denen die gängigen SELECT, INSERT und UPDATE Befehle umgesetzt werden können.

## 2.4. State Machines in Qt

Um entwickelte Zustandsautomaten zu implementieren, bietet Qt ein Konzept „State Machines“ mit dazugehörigen Klassen.

Dieses Konzept basiert darauf, einen, in beispielsweise UML vorliegenden, Zustandsgrafiken möglichst einfach zu implementieren.

Eine einfache Zustandsmaschine, dargestellt in Abbildung 2.2, kann wie folgt implementiert werden:

Zunächst wird die State Machine und ihre Zustände erzeugt (Listing 2.2).

Listing 2.2: State Machine Beispiel Teil 1

```
QStateMachine machine;
2   QState *s1 = new QState();
3   QState *s2 = new QState();
4   QState *s3 = new QState();
```

## 2. Grundlagen und Konzepte von Qt

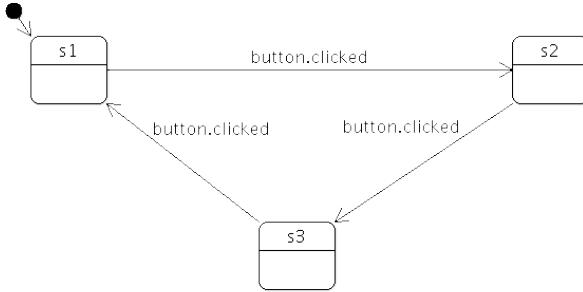


Abbildung 2.2.: State Machine Beispiel

Anschließend werden jedem Zustand alle für diesen Zustand vorhandenen Transitionen mit der `QState::addTransition()` Funktion hinzugefügt (Listing 2.3). Dabei wird angegeben, bei welchem Event (Signal) die Transition ausgeführt werden soll. Wenn ein Transitionssignal außerhalb des Zustands auftritt, wird die Transition nicht ausgeführt.

Listing 2.3: State Machine Beispiel Teil 2

```
2    s1->addTransition( button , SIGNAL( clicked() ) , s2 );
      s2->addTransition( button , SIGNAL( clicked() ) , s3 );
      s3->addTransition( button , SIGNAL( clicked() ) , s1 );
```

Listing 2.4 zeigt, wie anschließend die Zustände dem Zustandsautomaten hinzugefügt werden. Außerdem wird der initiale Zustand gesetzt, der bei Programmstart aktiv sein soll.

Listing 2.4: State Machine Beispiel Teil 3

```
2    machine.addState( s1 );
      machine.addState( s2 );
      machine.addState( s3 );
4    machine.setInitialState( s1 );
```

Abschließend muss die Zustandsmaschine gestartet werden (Listing 2.5). Ab diesem Zeitpunkt reagiert der Automat auf die Transitionen und wechselt zwischen den Zuständen.

Listing 2.5: State Machine Beispiel Teil 4

```
machine.start();
```

Hauptvorteil der so implementierten Zustandsmaschine ist, dass diese vollkommen asynchron zum restlichen Programm abläuft. Somit müssen keine zusätzlichen Schleifen in Tasks gestartet werden.

Um Funktionen mit einem Zustand zu verknüpfen kann ein Signal, welches jeder Zustand emittiert, genutzt werden. Bei dem Eingang in einen Zustand wird das

## *2. Grundlagen und Konzepte von Qt*

Signal entered() emittiert, bei dem Verlassen eines Zustands das Signal exited(). In dieser Arbeit wurde ausschließlich mit dem entered() Signal gearbeitet, da dies die Anforderungen ausreichend erfüllt hat.

Das aufgezeigte Beispiel und weitergehende Erklärungen zu State Machines und den weiteren Klassen sind in [Qt 19f] zu finden. Darunter sind auch Konzepte zu finden, die ermöglichen, Eigenschaften direkt an einen Zustand zu binden.

## **2.5. QListWidget**

Das QListWidget ermöglicht es, selbst programmierte QListWidgetItem in einer Liste zu organisieren und darzustellen. Zusätzlich werden diverse Funktionen bereitgestellt, mit denen sich die QListWidgetItem ordnen, ergänzen oder finden lassen (vgl. [Qt 19b]).

Über die Auswahl der allgemeinen Widgets kann das QListWidget im Layout-Manager ausgewählt werden.

# 3. Schnittstelle zum Robotino

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 3.1 dargestellte Vereinbarung wurde für die Ports getroffen.

Roboter IP Adresse	Sendeport	Empfangsport
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 3.1.: Portvereinbarung

## 3.1. 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 3.1, dargestellt.

Dem Sequenzdiagramm ist zu entnehmen, dass der Robotino nach Programmstart zyklisch Daten sendet. Die Zykluszeit beträgt 500 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 3.2 beschrieben. Solange der Robotino läuft werden die Daten gesendet. Dadurch kann auch ein Ausfall der Kommunikation festgestellt werden.

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 4.6.5 ist die in einer State-Machine implementierte Schleife beschrieben. Die Schleife kann durch

### 3. Schnittstelle zum Robotino

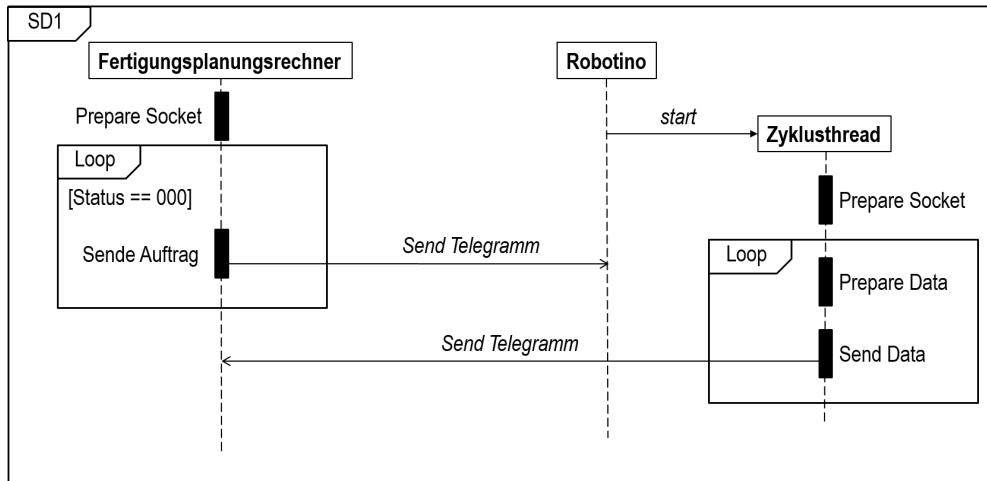


Abbildung 3.1.: Sequenzdiagramm

Empfangen einer Statusänderung des Robotinos verlassen werden. Dadurch wird eine erfolgreiche Übertragung des Auftrags sichergestellt werden. Die im Auftrag befindlichen Daten sind in Abschnitt 3.2 näher erläutert.

## 3.2. 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.

### 3.2.1. Telegramm vom Fertigungsplanungsrechner zu Gewerk 2

Der Auftrag, der an den Robotino gesendet wird, enthält 5 Bytes. Eine Aufschlüsselung ist in Tabelle 3.2 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 3.2.: Telegramm zu Gewerk 2

Das erste Byte klassifiziert die Auftragsart. Diese beschreibt, welche Tätigkeit der Robotino als nächstes tun soll. Auftragsart 1 bedeutet, dass der Robotino einen Transportauftrag erhalten hat, also ein Werkstück von einer Position zu

### 3. Schnittstelle zum Robotino

einer anderen Position 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.

Um Robotino 5 mit einzubinden wurde mit Gewerk 4 eine Sonderform des Protokolls vereinbart. Die Vereinbarung sieht vor, die Auftragsart zum Starten der Ladefahrt auf 1 zu setzen, wobei die Position 1 die 06 ist. Alle weiteren Felder sollen mit 00 gefüllt werden. Um die Ladefahrt erneut starten zu können, kann die Auftragsart auf 0 zurückgesetzt werden.

#### 3.2.2. Positions kodierung

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 3.2 dargestellt. Hier ist die Anordnung von Stationen, Parkplätzen und Ladestationen im Raum dargestellt, mitsamt ihrer Positions kodierung.

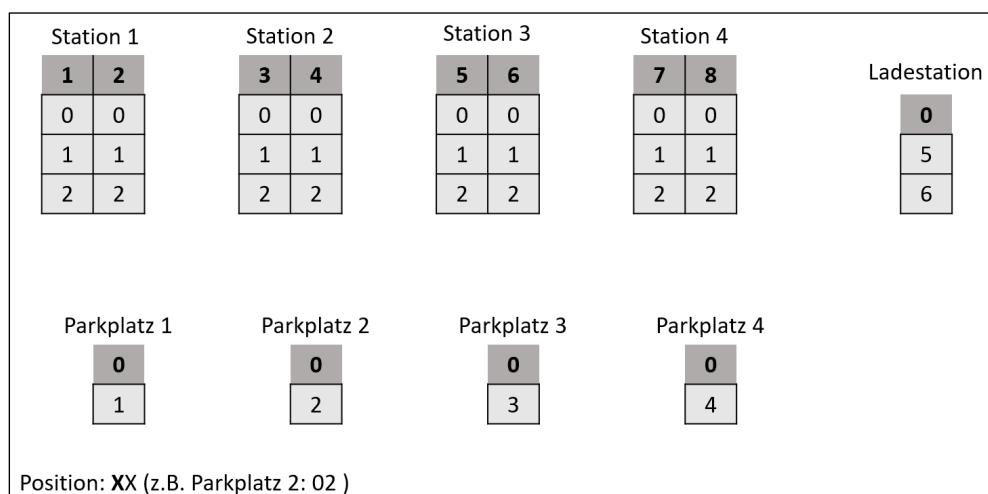


Abbildung 3.2.: Positions kodierung

### 3. Schnittstelle zum Robotino

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 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.

#### 3.2.3. 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 3.3 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 3.3.: Telegramm von Gewerk 2

Im ersten Byte sendet der Robotino einen Error. Die einzelnen Errortypen und die nötige Reaktion sind in Kapitel 3.3 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, dieses 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.

### *3. Schnittstelle zum Robotino*

Das vierte Byte wird von Gewerk 2 für eine interne Bekanntmachung verschiedener Roboterstatus 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 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 3.2.2 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.

## **3.3. Fehlertypen und Behebung**

Der Robotino kann über den Errorstatus verschiedene aufgetretene Fehler anzeigen. Drei dieser Fehlertypen und ihre Behandlung sind in den nächsten Abschnitten beschrieben. Im Normalfall ist der Error 1. Das bedeutet es ist alles in Ordnung.

### **Fehlertyp 2**

Wenn der Fehlertyp 2 empfangen wurde, bedeutet das, der Robotino hat sein Werkstück verloren. Diese Meldung tritt genau dann auf, wenn der Greifer des Robotinos zwar geschlossen ist, die Lichtschranke jedoch ein Verlust des Werkstücks detektiert.

Zur Behebung des Errors wird die Auftragsplanung pausiert und das verlorene Werkstück gesucht und aus der Fertigungsstraße entfernt. Anschließend wird der

### *3. Schnittstelle zum Robotino*

zugehörige Prozess geblockt. Die Zielstation des Robotinos wird per Hand freigegeben und eventuelle Reservierungen vom entfernten Werkstück aufgehoben.

In der Praxis konnte dieser Fehler nur mit Gewalt hervorgerufen werden, da ein geöffnetes Werkstück sehr stark festgehalten wird und ein Verlust damit unwahrscheinlich ist.

#### **Fehlertyp 3**

Wenn der Robotino einen Transportauftrag erhält und am Abholort kein Werkstück greifen kann, wird Fehlertyp 3 gesendet.

Die Fehlerbehebung ist identisch zur Fehlerbehebung von Fehlertyp 2. Es muss erneut geprüft werden, welcher Prozess abgearbeitet werden sollte, und alle diesem Prozess zugeordneten Arbeitsplätze, Stationen o.ä. freigegeben werden. Der Auftrag muss blockiert werden.

Da die Auftragsvergabe die Position jedes Werkstück kennt, kann dieser Fehlerfall in der Praxis nur sehr selten aufgrund einer größeren Störung oder menschlichem Versagen auftreten. Ein möglicher Grund kann sein, dass der Robotino den Arbeitsplatz verfehlt hat oder das Werkstück von einem Anwender unbeabsichtigt entfernt wurde.

#### **Fehlertyp 4**

Ein Fehler von Typ 4 bedeutet, dass keine Route planbar ist. In der Visualisierung muss je nach Situation bei Auftreten des Fehlers verschieden reagiert werden.

Wenn der Robotino direkt nach Erhalten eines Auftrags den Error sendet, kann der Auftrag direkt neu vergeben werden.

Für den Fall, dass der Robotino inmitten eines Transportvorgangs eine Route nicht berechnen kann, muss im schlimmsten Fall der Robotino defekt geschaltet werden. Im Normalfall sollte der Robotino nachdem der Error auftrat die Route in bestimmten Zeitabständen versuchen neu zu berechnen, daher kann der Error zunächst nur beobachtet werden. Sollte der Robotino neu gestartet werden, muss wie bei Fehlertyp 2 vorgegangen werden.

Wenn der Robotino ohne einen Transportauftrag keine Route planen kann, ist dies für die Auftragsplanung nur bedingt relevant und der Error kann als Info angesehen werden.

# 4. Implementierung

In diesem Kapitel geht es um die Implementierung des Programms in Qt. Dazu wird zunächst eine Übersicht über die Programmstruktur gegeben. Anschließend werden die wichtigsten Klassen näher beleuchtet und ihre Funktion erläutert.

## 4.1. Programmstruktur

In Abbildung 4.1 ist eine Übersicht der wichtigsten Klassen dargestellt.

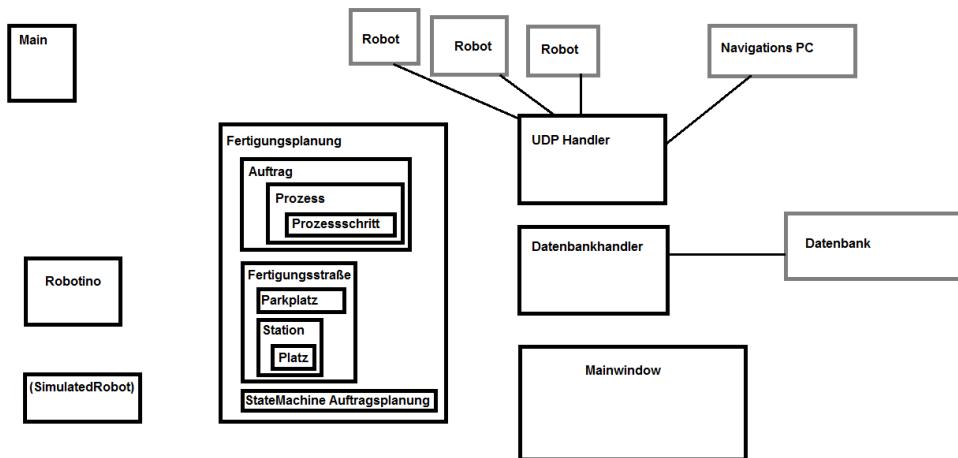


Abbildung 4.1.: Klassendiagramm

Die initiale Klasse bildet die Main, oben links. In der Main werden die anderen Klassen erzeugt, sowie Verbindungen zwischen den Klassen hergestellt.

Nach dem Ablauf der Main ist die Klasse Fertigungsplanung die kontinuierlich laufende Klasse des Programms. In der Fertigungsplanung findet die Auftragsvergabe statt. Für den Überblick über die Aufträge enthält die Fertigungsplanung eine Liste von Aufträgen. Jeder dieser Aufträge enthält eine Liste von Prozessen, welche wiederum eine Liste von Prozessschritten beinhalten.

Innerhalb der Fertigungsplanung wird die Fertigungsstraße in einer Klasse abgebildet. Diese besteht wiederum aus Parkplätzen und Stationen. Jede Station enthält einzelne Arbeitsplätze.

Mit der in der Fertigungsplanung implementierten State-Machine wird die Auftragsplanung durchgeführt.

#### *4. Implementierung*

Auf der linken Seite der Abbildung ist die Robotino Klasse dargestellt. Diese enthält alle Stati und Informationen zu den Robotinos. Je real existierendem Robotino wird ein Objekt dieser Klasse erzeugt.

Im Simulationsbetrieb wird die Klasse SimulatedRobot genutzt, um die Statusänderungen der Robotinos nachzubilden. Diese Klasse wird mit dem UDP-Handler verbunden.

Der UDP-Handler ist die Schnittstelle (Interface) zwischen dem Programm und den Robotinos, sowie dem Navigations-PC. Die Klasse stellt Funktionen zum Datenaustausch und zur ersten Datenauswertung zur Verfügung.

Ähnlich dem UDP-Handler sorgt die Klasse Datenbankhandler für die Bereitstellung von Funktionen, um auf die Datenbank zugreifen zu können. Die initiale Verbindung zur Datenbank erfolgt ebenfalls über diese Klasse.

In der Klasse Mainwindow erfolgt die Visualisierung und Verknüpfung zur grafischen Oberfläche. Diese Klasse ist im Kapitel 5 näher beleuchtet.

Neben den genannten Klassen existieren noch weitere kleinere Klassen, um die Aufgezeigten zu verwalten oder zu verbinden.

## **4.2. Database-Handler**

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

### **4.2.1. 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 4.1, des Datenbank-Handler wird zunächst eine QSqlDatabase (siehe Kapitel 2.3) 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 10ff dargestellt, abgefangen und angezeigt werden.

#### 4. Implementierung

Listing 4.1: Databank-Handler Konstruktor

```
1 db = QSqlDatabase :: addDatabase ("QMYSQL") ;
2 db . setHostName ("192.168.0.107") ;
3 db . setPort (3306) ;
4 db . setDatabaseName ("vpj") ;
5 db . setUserName ("FP") ;
6 db . setPassword ("1234") ;
7
8 bool ok = db . open () ;
9
10 QSqlError err ;
11 if (err . type () != QSqlError :: NoError ){
12     qCritical () << err . text () ;
13 }
```

### Initialisierung

Um die Datenbank zu Programmstart auf einen definierten Anfangszustand zu bringen, wurde eine Initialisierungsfunktion (Listing 4.2) geschrieben. Darin werden alle Zeitstempel, Aufträge und Prozesse (Werkstücke) gelöscht.

Weiterhin wird der Roboter in der Datenbank mit Status 0 initialisiert 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 4.2: Databank-Handler Initialisierung

```
1 SendOverQuary (query , "DELETE FROM vpj . rfid _ timestamp ;" ) ;
2 SendOverQuary (query , "UPDATE vpj . roboter SET
3     betriebsstatus=0, Werkstueck _ RFID _ Werkstueck=NULL;" ) ;
4 SendOverQuary (query , "UPDATE vpj . parkplatz SET Status=0,
5     Roboter _ id _ Roboter=NULL;" ) ;
6 SendOverQuary (query , "UPDATE vpj . arbeitsplatz SET Status
7     =0, Roboter _ id _ Roboter=NULL, Werkstueck _ RFID _ Werkstueck
8     =NULL;" ) ;
9 SendOverQuary (query , "UPDATE vpj . taggen SET
10     Werkstueck _ RFID _ Werkstueck=NULL WHERE id _ taggen=1;" ) ;
11 SendOverQuary (query , "DELETE FROM vpj . werkstueck ;" ) ;
12 SendOverQuary (query , "DELETE FROM vpj . auftrag ;" ) ;
```

## 4. Implementierung

### Allgemeine Sendefunktion

Jede Kommunikation mit der Datenbank beginnt mit der Erzeugung einer Query, in Listing 4.3 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 4.3: Databank-Handler Senden über Query

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

### 4.2.2. Getter

Um auf Daten in der Datenbank zuzugreifen, wurden Funktionen bereitgestellt, die diese zurückgeben. So kann über die Getter festgestellt werden, ob sich ein Werkstück im Lager befindet oder ob ein Roboter oder eine Station frei oder reserviert ist. Eine Funktion liefert alle in der Datenbank existierenden Prozesse zurück.

Es kann der nächste freie Parkplatz oder die nächste freie Ladestation zurückgegeben werden. Bei Übergabe einer Station kann das zugehörige Werkstück oder andersherum bei Übergabe des Werkstücks die zugehörige Station und zugehöriger Robotino gefunden werden.

Es kann einfach festgestellt werden, ob sich ein Robotino aktuell auf einem Parkplatz befindet.

Zuletzt werden über zwei Funktionen die Timestamps von Werkstück oder Station bereitgestellt.

Nähere Informationen sind in Kapitel 8.7.1 zu finden.

### 4.2.3. Updates

Um die Daten der Datenbank zu aktualisieren wurden weitere Funktionen bereitgestellt. Mit den Update-Funktionen kann der Arbeitsplatz-Status oder der Prozessfortschritt aktualisiert werden.

Die Robotino-Akkustände sowie deren Stati können auch aktualisiert werden.

Dem Arbeitsplatz kann außerdem ein Werkstück oder Robotino zugewiesen, oder entfernt, werden.

#### 4. Implementierung

Über eine Update-Funktion wird der Schreib-Tag, der die RFIDs beinhaltet aktualisiert.

Das Kapitel 8.7.2 beschreibt den Inhalt der Funktionen genauer.

#### 4.2.4. Setter

Über Setter-Funktionen können Einträge in der Datenbank gesetzt werden. Die beiden Hauptfunktionen hier setzen die erzeugten Aufträge oder erzeugten Prozesse.

Innerhalb der SendProzess-Funktion wird unterschieden, ob die Datenbank den Prozess bereits beinhaltet, dieser also nur aktualisiert, oder neu erzeugt werden muss.

Die Implementierung der Setter-Funktionen ist in Kapitel 8.7.3 dargestellt.

### 4.3. 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.

#### 4.3.1. UDP-Handler für Navigationsrechner

Um auf Daten des Navigationsrechners zu reagieren wird im Konstruktor des HDP-Handler (vgl. Listing 4.4) ein QUdpSocket (vgl. Kapitel 2.2) 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 4.4: UDP-Handler Konstruktor für Navigationsdatenaustausch

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

#### 4. Implementierung

Die Funktion `readPendingNavigationData()`, dargestellt in Listing 4.5, 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 4.5: Daten über UDP empfangen

```

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

```

Das empfangene Datagramm wird zunächst in der Funktion `processTheNavigationData` (Listing 4.6 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 4.6 Zeile 16ff) übergeben.

Die Rohdaten aus dem Datagramm müssen für jedes Element dekodiert werden. Dazu wird in Zeile 10 aus dem Datagramm ein QByteArray 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 QByteArrays werden in Zeile 11 interpretiert als ein Element vom Typ Double.

Listing 4.6: UDP-Handler Navigationsdatenaustausch

```

1   if (datagram.data().length() != 448)
2   {
3       qDebug() << "Received Navigation Data have wrong
4           dimensions: " << datagram.data().length();
5   }
6   else
7   {
8       QVector<double> vektor(56);
9       for (int i = 0; i < 56; i++)
10      {
11          QByteArray temp = datagram.data().mid(i*8,8);
12          vektor[i] = *reinterpret_cast<const double*>(temp.
13              data());
14      }
15      prepareNavigationDataforRobot(vektor);
16  }
17  QVector<int> posr1(3);

```

#### 4. Implementierung

```

18 | posr1[0] = static_cast<int>(values[0]);
19 | posr1[1] = static_cast<int>(values[1]);
20 | posr1[2] = static_cast<int>(qRadiansToDegrees(values[2]));
21 | [...]
22 | emit NavigationDataReceivedR1(posr1);

```

Die entstandenen Doublewerte im übergebenen Vektor werden in der Funktion `prepareNavigationDataforRobot()` ausgewertet. Zeile 16ff 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.

#### 4.3.2. UDP-Handler für Robotino

Der überladene Konstruktor, um die UDP-Kommunikation zu einem Robotino aufzubauen, ist in Listing 4.7 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 6) einzubinden.

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 QUpdSocket an den Receive-Port gebunden und der Nachrichtenempfang mit der Funktion `readPendingRobotData()` verknüpft.

Listing 4.7: UDP-Handler Konstruktor für Navigationsdatenaustausch

```

UdpHandler::UdpHandler(bool simulated, QHostAddress ip,
    quint16 sport, quint16 rport, QObject *parent) :
    QObject(parent)
{
    this->ip = ip;
    this->receivingport = rport;
    this->sendingport = sport;
    this->simulated = simulated;
    lastSendValues = QVector<double>(5);

    if (!simulated)
    {
        socket = new QUpdSocket(this);
        socket->bind(rport, QUpdSocket::ShareAddress);
        connect(socket, SIGNAL(readyRead()), this, SLOT(
            readPendingRobotData()));
    }
    else

```

#### 4. Implementierung

```

16     {
17         r . Start () ;
18         QObject :: connect (&r , &SimulatedRobot :: 
19             StatusDataSimulatedReceived , this , &UdpHandler
20                 :: simulatedRobotDataReceived ) ;
21     }
22 }
```

Zusätzlich zu den Lesefunktionen, die Daten über UDP empfangen, gibt es für die Robotinos eine Schreibfunktion WriteData() (Listing 4.8), 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 geschickt.

Listing 4.8: Daten über UDP verschicken

```

lastSendValues = values ;
2 if ( simulated )
3 {
4     r . auftragsart = static _ cast < int > ( values [ 0 ] ) ;
5     r . pos1 = static _ cast < int > ( values [ 1 ] ) ;
6     r . pos2 = static _ cast < int > ( values [ 2 ] ) ;
7 }
8 else
9 {
10    QByteArray Data ;
11    for ( int i = 0 ; i < 5 ; i ++ )
12    {
13        Data . append ( QByteArray :: fromRawData (
14            reinterpret _ cast < char * > ( & values [ i ] ) , sizeof (
15                values [ i ] ) ) ;
16    }
17    socket -> writeDatagram ( Data , ip , sendingport ) ;
18 }
```

Die Funktion readPendingRobotData gleicht der bereits im UDP-Handler für Navigationsdaten beschriebenen Funktion (Listing 4.5).

Die nachfolgend aufgerufene Funktion processTheRobotData ist, bis auf die Längen der Vektoren und des Datagramms, gleich der zuvor in Listing 4.6 gezeigten.

## 4. Implementierung

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.

## 4.4. 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 und eine eindeutige RoboterID. Weiterhin werden alle vom Robotino gesendeten Daten wie Status, Position, Error, Hindernis, Akku und Greifer gespeichert (vgl. 3).

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 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.

### 4.4.1. 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 Robotererrortypen (vgl. Kapitel 3) 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.

#### 4.4.2. 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 ignoriert und der Robotino wird 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.

### 4.5. 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.

#### 4.5.1. Prozessschritt

Ein Prozessschritt besteht aus genau einer Stations-ID verknüpft mit einer festen Bearbeitungsdauer in Sekunden. Dies beschreibt 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.

## *4. Implementierung*

### **4.5.2. 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.

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 5.5.4) 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.

Um auf den richtigen Prozessschritt innerhalb der Liste zuzugreifen sind zwei unterschiedliche Funktionen implementiert. Beide Funktionen unterscheiden sich ausschließlich in ihrem Rückgabewert. In Abbildung 4.2 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

#### 4. Implementierung

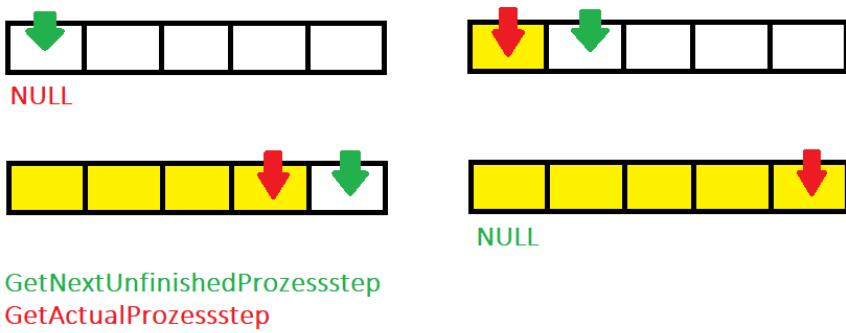


Abbildung 4.2.: Visualisierung des Rückgabewertes der Funktionen GetNextUnfinishedProzessstep() und GetActualProzessstep()

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.

Die Funktion GetActualProzessstep() wird in der Fertigungsplanung zur Planung der Aufträge verwendet (siehe 4.6.5). GetNextUnfinishedProzessstep() findet Anwendung sobald auf einem aktuellen Prozess eine Änderung auftritt oder umgesetzt werden soll. Wenn beispielsweise der Timer abgelaufen ist, wird über die Funktion TimerElapsed() der zurückgegebene Prozessschritt als fertig markiert.

### 4.5.3. 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 Prozessfortschritt hervorruft. Dies wiederum lässt den entsprechenden Auftragsfortschritt aktualisieren (siehe Listing 4.9).

Ein bestimmter Prozess aus der Liste kann über die Funktion GetProzessByID() anhand seiner eindeutigen 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 GetNextUnfinished-

#### 4. Implementierung

Prozessstep() aus dem Prozess, eine Liste aller entsprechenden Prozessschritte zurück (vgl. Abschnitt 4.5.2). 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 4.6.5).

## 4.6. Fertigungsplanung

Die Klasse Fertigungsplanung enthält alle verfügbaren Roboter, die Datenbankschnittstelle, die abzuarbeitenden Aufträge (vgl. Abschnitt 4.5), 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 5.5.4) angestoßen werden können.

In der Fertigungsplanungsklasse werden somit alle Funktionalitäten gebündelt, die nicht direkt mit der Visualisierung zusammenhängen.

### 4.6.1. 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 4.6.1) 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 5.9).

Die Prozessliste kann über die Funktion AddProzess() dynamisch, während das Programm läuft, erweitert werden. Dies geschieht bei Absenden eines eingestelltes Prozesses in der Visualisierung (5.7).

### Aufträge hinzufügen

Durch Einstellen und Absenden eines Auftrags in der Visualisierung (5.8) 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 als Prozesskopie erzeugt. Den so kopierten Prozessen

#### 4. Implementierung

werden alle zugehörigen Prozessschritte als Kopie angehängt, um eigenständige Aufträge zu erhalten. Zuletzt werden die Prozesse und Aufträge miteinander verknüpft (siehe Listing 4.9).

Listing 4.9: 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  
    {  
        Prozess *p = new Prozess();  
        [...] // Prozess + Prozessschritte kopieren  
        A->Prozesse.append(p);  
10      QObject::connect(p, &Prozess::FortschrittUpdated,  
                           A, &Auftrag::UpdateFortschritt);  
      QObject::connect(p, &Prozess::ProzessFinished,  
                       this, &Fertigungsplanung::StationReady);  
12    }  
}  
}
```

Abschließend wird der Auftrag der Auftragsliste angehängt, an die Datenbank gesendet und ein Auftragsitem in der Visualisierung erzeugt (Kapitel 5.4).

### Standardprozesse

Folgend werden die in Abschnitt 4.6.1 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 4.3 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.

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 wurden so angepasst, dass alle Stationen möglichst gleichmäßig häufig durchfahren werden. Weiterhin kann aus dem Diagramm abgelesen werden, ob durch ungünstige Auftragsplanung ein Deadlock angefahren werden kann, also alle Aufträge irgendwann durch andere Aufträge

#### 4. Implementierung

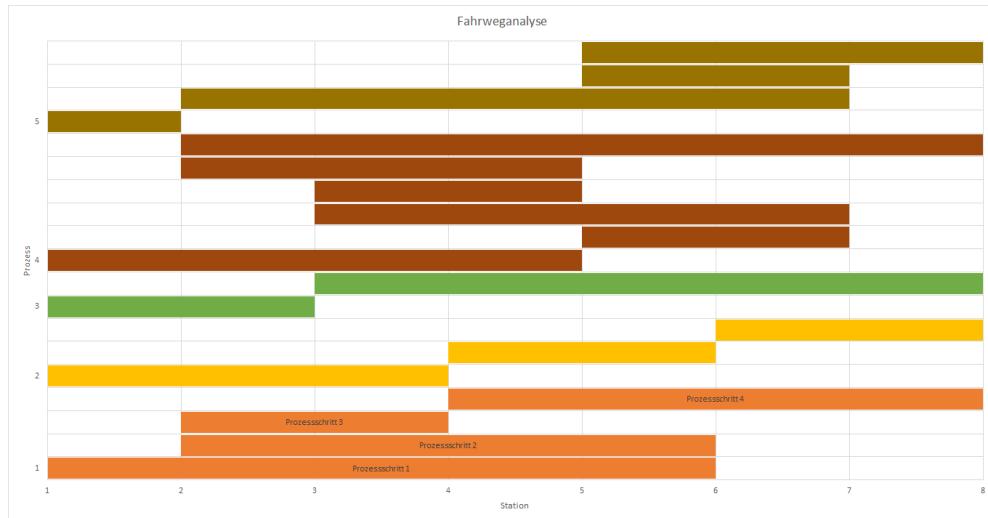


Abbildung 4.3.: Diagramm Fahrweganalyse

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 4.4 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.

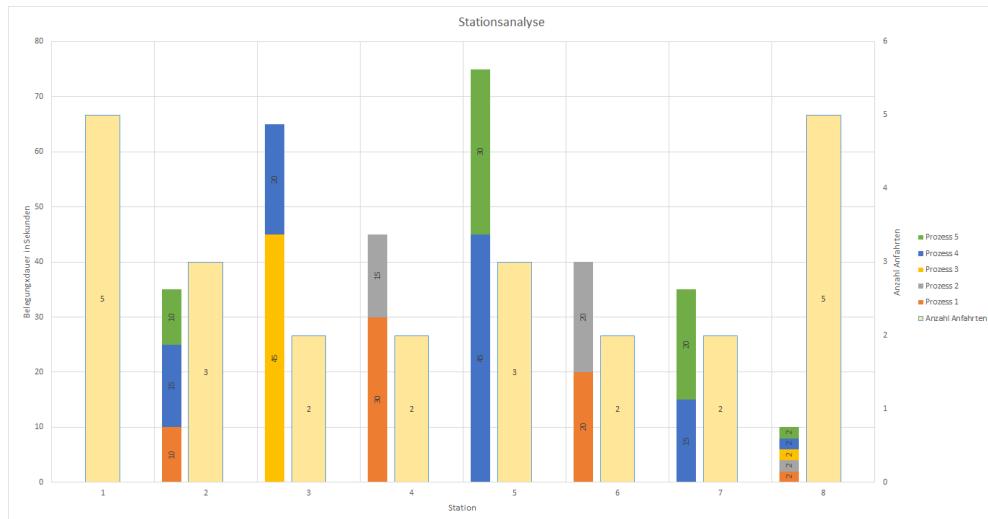


Abbildung 4.4.: Diagramm Stationsanalyse

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 zweimal angefahren. So wird sichergestellt, dass die Stationen möglichst gleichmäßig belastet werden.

Die farblich unterschiedlichen Säulen zeigen die kürzeste Belegungsdauer der Ar-

#### 4. Implementierung

beitsplätze an. Die Balken ergeben sich aus der Dauer eines einzelnen Prozessschrittes an einem Arbeitsplatz. Es ist zu erkennen, dass die Belegungsdauer für alle Prozesse zwischen 35 und 75 Sekunden, je Station liegt. Die 2 Sekunden Bearbeitungsdauer 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 zwar die Gesamtzeit in Station 5 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. Gegensätzlich dazu ist die Bearbeitungszeit an Station 23 von Prozess 3 mit 45 Sekunden sehr hoch angesetzt, da dieser Prozess keine weitere Station anfährt.

Es wurde versucht, die durchschnittliche Belegungsdauer der Stationen anzuleichen, um alle Arbeitsplätze möglichst homogen auszulasten.

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

#### 4.6.2. Roboter Statusänderungen

Wenn eine der, in der Fertigungsplanung enthalten, Roboterklassen ein Event feuert, welches eine Statusänderung beinhaltet (vgl. Kapitel 4.3), wird über Funktionen in der Fertigungsplanung auf diese reagiert. So wird der Greifer oder das Statusflag eines Roboters bei einer Änderung aktualisiert. Beim Empfang eines Errorevents des Roboters wird je nach Errortyp 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 Roboters auf eins geändert haben, also der Roboter auf dem Weg zu etwas sein, 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.

## *4. Implementierung*

### **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 5.9) auf einen leeren String gesetzt.

Bei dem 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 der Benutzer die Kontrolle über das Gesamtsystem behalten sollte. Somit muss je nach Fehlertyp ein unterschiedlicher Lösungsweg verfolgt werden.

Die auftretenden Fehler und der Lösungsweg ist in Kapitel 3.3 zu finden.

### **4.6.3. Hard-Code Funktionen**

Die mit der Hard-Code Area erzeugten Befehle aus der Visualisierung (vgl. Kapitel 5.5.4) 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.

#### 4. Implementierung

##### 4.6.4. Zustandsdiagramm

Das Zustandsdiagramm in Abbildung 4.5 beschreibt die Logik der Auftragsvergabe an die Roboter. Das Diagramm wurde anschließend als State Machine (vgl. 2.4) 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 4.6.5 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.

##### 4.6.5. 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. 4.7.1).

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 4.10 abgebildet.

Listing 4.10: Start und Initialisierung der State Machine

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

##### Zustand CheckRobots

Im Initialzustand der State Machine, CheckRobots, werden alle vier Roboter nacheinander auf Bereitschaft geprüft. Die Reihenfolge der zu überprüfenden Ro-

#### 4. Implementierung

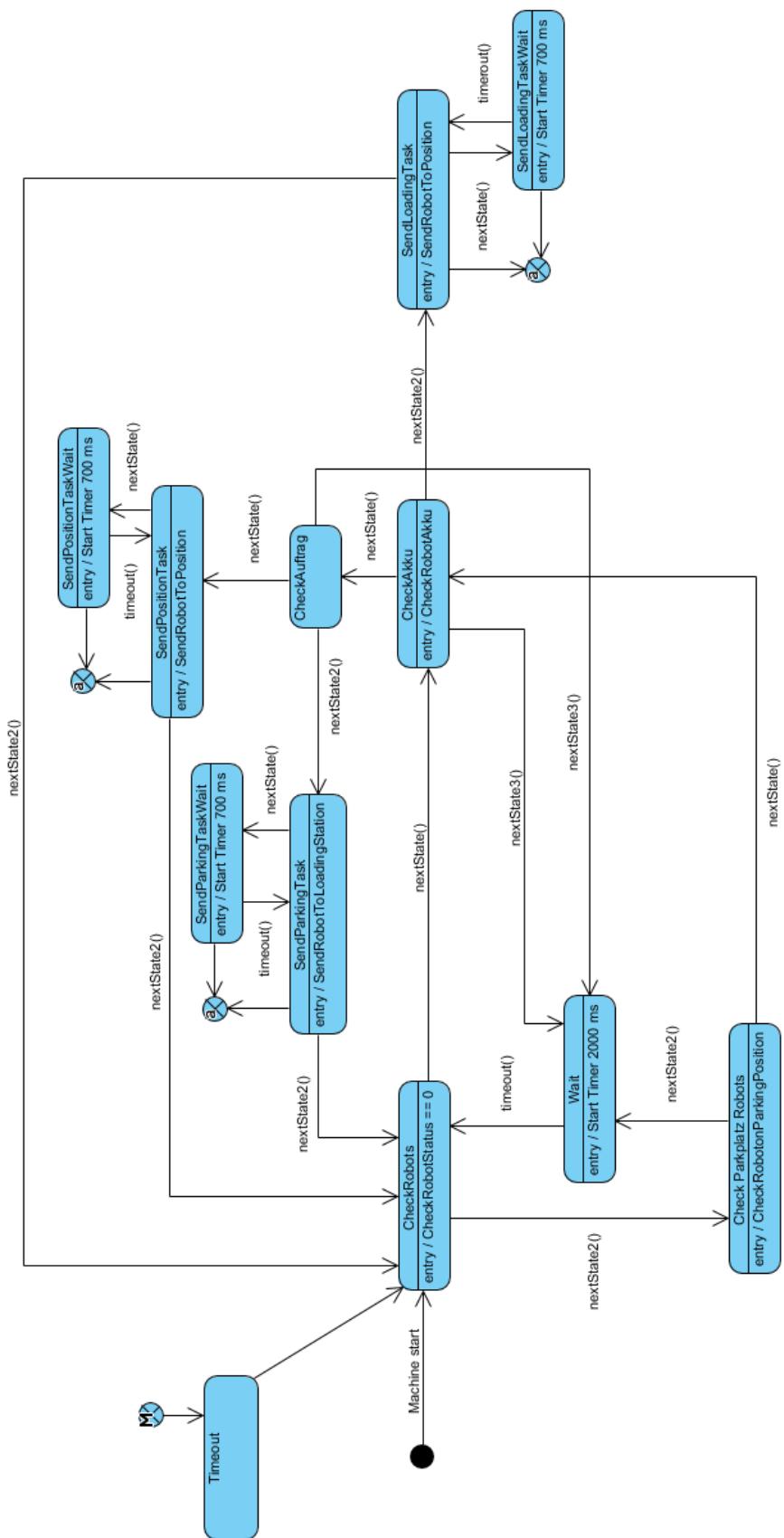


Abbildung 4.5.: Zustandsdiagramm der Auftragsplanung

## *4. Implementierung*

boter wird dabei durch einen hierfür entwickelten Shuffle-Algorithmus (4.6.6) 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.

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 vermieden, 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) keine Endlos-schleifen vorgesehen sind, wurde im Zustand „Wait“ mittels Timer die Zustandsmaschine künstlich verlangsamt. Ein Entfernen des Zustands führt zum Programmburstz, 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.

## 4. Implementierung

### 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 3.1). Es wird nach jedem Sendevorgang ein kurzer Wartezustand aufgerufen, der, gleich dem Zustand Wait, einen Programmabsturz verhindert. 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 dadurch 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 Zielerbeitsplatz der genutzte Roboter

#### *4. Implementierung*

zugeordnet. Am Startarbeitsplatz wird das vorhandene Werkstück in der Datenbank entfernt und dem Zielarbeitsplatz 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 Zielarbeitsplatz reserviert. Schlussendlich wird in der Visualisierung der Tooltip (vgl. Kapitel 5.9) des Start- und Zielarbeitsplatzes 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 4.5). 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.

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.

#### **4.6.6. 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.

#### 4. Implementierung

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 4.11 Zeile 1-3).

Listing 4.11: Shuffle-Algorithmus

```

1   QList<int> liste;
2   int i = QRandomGenerator::global()->bounded(1, 5);
3   liste.append(i);
4   i = QRandomGenerator::global()->bounded(1, 4);
5   (liste.contains(i)) ? liste.append(i+1) : liste.append
6   (i);
7   i = QRandomGenerator::global()->bounded(1, 3);
8   if (liste.contains(i))
9   {
10      if (liste.contains(i+1))
11      {
12          liste.append(i+2);
13      }
14      else
15      {
16          liste.append(i+1);
17      }
18  }
19  else
20  {
21      liste.append(i);
22  }
23
24  if (!liste.contains(1)) {liste.append(1);}
25  else if (!liste.contains(2)) {liste.append(2);}
26  else if (!liste.contains(3)) {liste.append(3);}
27  else if (!liste.contains(4)) {liste.append(4);}
28  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 4.11 Zeile 4f).

Um die nächste Zahl hinzuzufügen wird im dritten Schritt (Listing 4.11 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.

#### 4. Implementierung

Zuletzt wird die noch fehlende Zahl der Liste ergänzt (Listing 4.11 Zeile 22ff) und die Liste zurückgegeben.

## 4.7. Weitere Klassen

Neben den bisher ausführlicher beschriebenen Klassen, gibt es noch weitere Klassen, die für Organisation, Strukturierung oder als Hilfe für bestimmte Teilgebiete dienen.

In den folgenden Abschnitten sind werden die wichtigsten unter ihnen kurz beleuchtet.

### 4.7.1. Main

Die initial aufgerufene Klasse des Programms ist die Main-Funktion. In dieser wird die Applikation, also das Anzeigefenster samt Logik, gestartet. Der Hauptteil der Funktion verbindet auftretende Signale mit zugehörigen Slots.

Über eine Property kann in der Main festgelegt werden, ob das Programm im Simulationsmodus gestartet werden soll, oder im realen Betrieb.

Listing 4.12 zeigt einige Funktionen, die folgend erwähnt werden. Diese sind aus dem Kontext genommen und stellen nur eine möglichst repräsentative Auswahl mit Reihenfolge der Main Klasse dar.

Listing 4.12: Main Funktion Teil 1

```
1 Robotino robot1(1, QHostAddress("192.168.0.11"), 25010,  
2   25011);  
3 UdpHandler robot1UDP(simulated, robot1.IP(), robot1.  
4   PortSend(), robot1.PortReceive());  
5 UdpHandler navUDP(25000);  
6 QObject ::connect(&robot1UDP, &UdpHandler ::  
7   StatusDataReceived, &robot1, &Robotino :: UpdateValues);  
8 QObject ::connect(&navUDP, &UdpHandler ::  
9   NavigationDataReceivedR1, &robot1, &Robotino ::  
10  UpdatePosition);  
11 QObject ::connect(&robot1, &Robotino :: PositionChanged, &w,  
12  &MainWindow :: UpdateRobotino1Position);  
13 QObject ::connect(&robot1, &Robotino :: AkkuChanged, &w, &  
14  MainWindow :: UpdateRobotino1Akku);  
15 QObject ::connect(&robot1, &Robotino :: GetUdpResponse, &  
16  robot1UDP, &UdpHandler :: sendRobotAlive);  
17 QObject ::connect(&robot1, &Robotino :: RobotAliveChanged, &w  
18  , &MainWindow :: UpdateRobotino1Alive);  
19 QObject ::connect(&robot1, &Robotino :: GreiferChanged, &w, &  
20  MainWindow :: UpdateRobotino1Greifer);
```

#### 4. Implementierung

```
12 | QObject :: connect(&robot1 , &Robotino :: HindernisChanged , &w,
    &MainWindow :: UpdateRobotino1Hindernis );
```

Zuerst werden Objekte der Roboterklassen (Zeile 1) erzeugt. Dies geschieht für alle Robotinos. Zugehörig werden alle UdpHandler instanziert (Zeile 3), welche die Kommunikation zu den Robotinos ermöglichen. Für die Kommunikation zum Navigationsrechner wird der notwendige UDPHandler in Zeile 3 instanziert.

Über die connect() Funktion werden die UDP-Handler und Robotino verbunden um auf Statusänderungen (Zeile 4) und Positionsänderungen (Zeile 5) zu reagieren. In den Zeilen 7ff dargestellt sind die Connect-Funktionen um die Visualisierung innerhalb der MainWindow-Klasse zu aktualisieren, und die Statusänderungen des Robotinos korrekt zu behandeln.

Listing 4.13 zeigt die anschließende Initialisierung (Zeile 1) der Fertigungsplanung und zugehörige Connect() Funktionen.

Listing 4.13: Main Funktion Teil 2 - Fertigungsplanung

```
1 Fertigungsplanung Fertigung ;
2 QObject :: connect(&w, &MainWindow :: NeuerAuftrag , &Fertigung
    , &Fertigungsplanung :: AddAuftrag );
3 QObject :: connect(&w, &MainWindow :: NeuerProzess , &Fertigung
    , &Fertigungsplanung :: AddProzess );
4 QObject :: connect(&Fertigung , &Fertigungsplanung :::
    DebugAreaString , &w, &MainWindow :: WriteToDebugTextArea )
    ;
5 QObject :: connect(&Fertigung . Fs . S1 . P1 , &Platz :: PlatzChanged
    , &w, &MainWindow :: UpdateStationsplatz );
6 QObject :: connect(&Fertigung . Fs . P1 , &Parkplatz :::
    PlatzChanged , &w, &MainWindow :: UpdateParkplatz );
```

Die Fertigungsplanung wird mit den Funktionen der Auftrags und Prozesserzeugung des MainWindow (Zeile 2 und 3), als auch dem Log-View (Zeile 4) verknüpft.

Anschließend werden alle Arbeits- und Parkplätze mit der zugehörigen Updatefunktion in der Visualisierung verbunden.

In weiteren nicht dargestellten Connect-Aufrufen werden die Funktionen der Robotinos mit denen der Fertigungsplanung verknüpft. Anschließend werden alle anklickbaren Elemente der Visualisierung mit den zugehörigen Funktionen verbunden.

Listing 4.14: Main Funktion Teil 3 - Datenbank und StateMachine

```
1 DatabaseHandler dbhandler ;
2 dbhandler . InitDB () ;
3
4 Fertigung . r1 = &robot1 ;
5 Fertigung . r1UDP = &robot1UDP ;
6 Fertigung . db = &dbhandler ;
```

#### 4. Implementierung

```
7 Fertigung . InitProzesses () ;  
8 Fertigung . AddTransitionsToStates () ;  
9 Fertigung . AddStatesToStateMachine () ;  
10 QObject :: connect (Fertigung . StateCheckRobots , &QState ::  
    entered , &Fertigung , &Fertigungsplanung :: CheckRobots ) ;  
11 Fertigung . Start () ;
```

Nachdem so alle UDP- und internen Verbindungen hergestellt wurden, wird die Datenbank initialisiert und die Verbindung hergestellt (Listing 4.14 Zeile 1f).

Alle Verknüpfungen zwischen Fertigungsplanung, DatenbankHandler und Visualisierung werden nachfolgend erzeugt. Das Konzept entspricht dem oben beschriebenen.

Der Fertigungsplanung werden nun alle erzeugten Objekte der Robotinos und aller Handler gesetzt (Zeilen 4 - 6).

Zuletzt wird die Zustandsmaschine der Fertigungsplanung vorbereitet. Dies geschieht in Zeilen 7ff. Besonders hervorzuheben ist hierbei, dass jeder Zustand des Zustandsautomaten eine eigene Funktion besitzt, die beim entered()-Event aufgerufen wird. So ruft beispielsweise das Betreten des Zustands StateCheckRobots (Zeile 10) die Funktion CheckRobots auf.

Abschließend wird die Zustandsmaschine in Zeile 11 gestartet und die Main Funktion wird geschlossen.

#### 4.7.2. Timestamp

Um die in der Datenbank erzeugten Timestamps zwischenzuspeichern und anzuzeigen wurde eine Timestamp Klasse entworfen.

Die Klasse enthält, zusätzlich zu dem Timestamp selbst, den zugehörigen Platz, das Werkstück und einen Status. Die Properties werden einmalig über den Konstruktor gesetzt.

Die Klasse stellt weiterhin eine Funktion bereit, die es ermöglicht, aus den so vorliegenden Daten einen String zu generieren. Dazu wird ein QDateTime Objekt erzeugt, dessen Zeit auf LocalTime gesetzt wird. Mit diesem kann nachfolgend ein einheitlicher String erzeugt werden.

#### 4.7.3. Fertigungsstrasse

Um auf Events innerhalb der Prozesskette zu reagieren wurde zusätzlich zu der in der Datenbank abgelegten Fertigungsstrasse, diese in Qt abgelegt. Beide werden redundant gepflegt und in der Auftragsvergabe berücksichtigt.

Die Fertigungsstrasse stellt Funktionen zur Verfügung, die ermöglichen, einen Arbeitsplatz oder eine Station anhand ihrer ID zurückzugeben. Außerdem sind alle acht Stationen, Parkplätze und Ladestationen hier in eingebetteten Klassen abgebildet.

#### *4. Implementierung*

Die Klasse der Stationen enthält neben zwei Arbeitsplätzen eine ID.

In die Klasse der Arbeitsplätze wurden neben Status und ID die Signale implementiert, die eine Arbeitsplatzänderung anzeigen. Es existieren weiterhin Funktionen um den Status des Arbeitsplatz zu beeinflussen.

#### **4.7.4. Auftrags- und Prozessitem**

Die in der Auftragsübersicht (siehe 5.4) dargestellten Items werden in zwei unterschiedlichen Klassen implementiert.

Auf der einen Seite gibt es die Klasse Prozessitem, die für die Visualisierung eines Prozesses benötigt wird. Ergänzend dazu existiert eine Klasse Auftragsitem, welche die Darstellung eines Auftrags in der Auftragsübersicht ermöglicht. Die beiden Klassen unterscheiden sich neben der Behandlung von Aufträgen oder Prozessen darin, dass im Prozessitem Funktionen bereitgestellt werden, mit denen ein Prozessitem geblockt werden kann.

Beide der Klassen stellen für sich eine vollständige Implementierung einer Visualisierung dar, inklusive Connect und Updatefunktionen. Im Rahmen der Dokumentation wird ausschließlich die weitaus umfassendere MainWindow Klasse genauer beschrieben, da diese alle Funktionalitäten beider ItemKlassen abdeckt.

# 5. 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.

## 5.1. 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 5.1.: VPJ-Logo

Das entworfene VPJ-Logo (Abb. 5.1) 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.

## 5. Visualisierung

### 5.2. Struktur

Die Visualisierung lässt sich in insgesamt 6 Bereiche unterteilen. Die Bereiche sind in Abbildung 5.2 dargestellt. Bereich I oben links zeigt die Fertigungsstraße mitsamt Robotern, Parkplätzen und Stationen. Dieser Bereich wird folgend Live-View genannt.

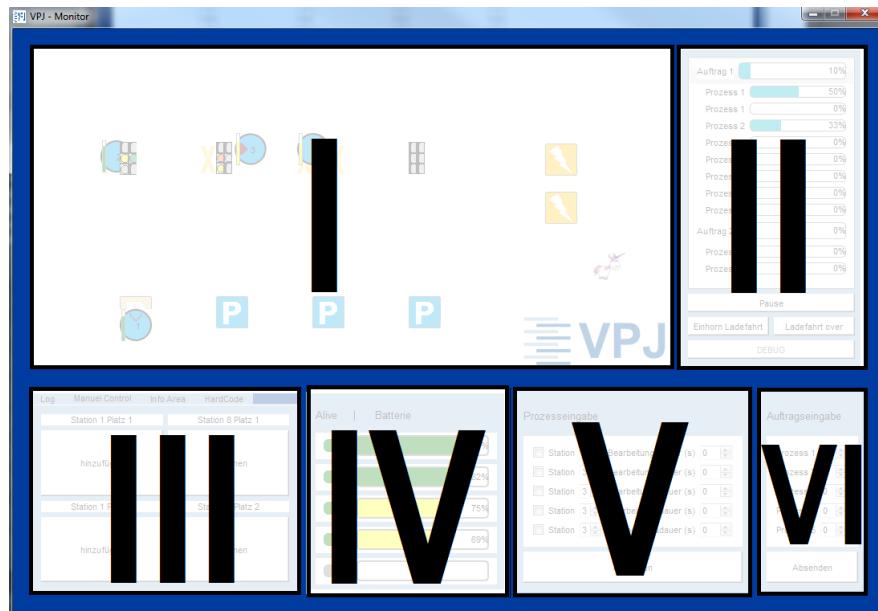


Abbildung 5.2.: Übersicht über die Bereiche innerhalb der Visualisierung

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

Innerhalb des Bereichs 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 5.3 ist eine Übersicht der Visualisierung im laufenden Betrieb dargestellt.

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

### 5.3. Live-View

Der Größte der Bereiche beinhaltet eine Live-View der Fertigungsstraße, dargestellt in Abbildung 5.4. Darin sind alle Stationen, die Robotinos, Parkplätze

## 5. Visualisierung

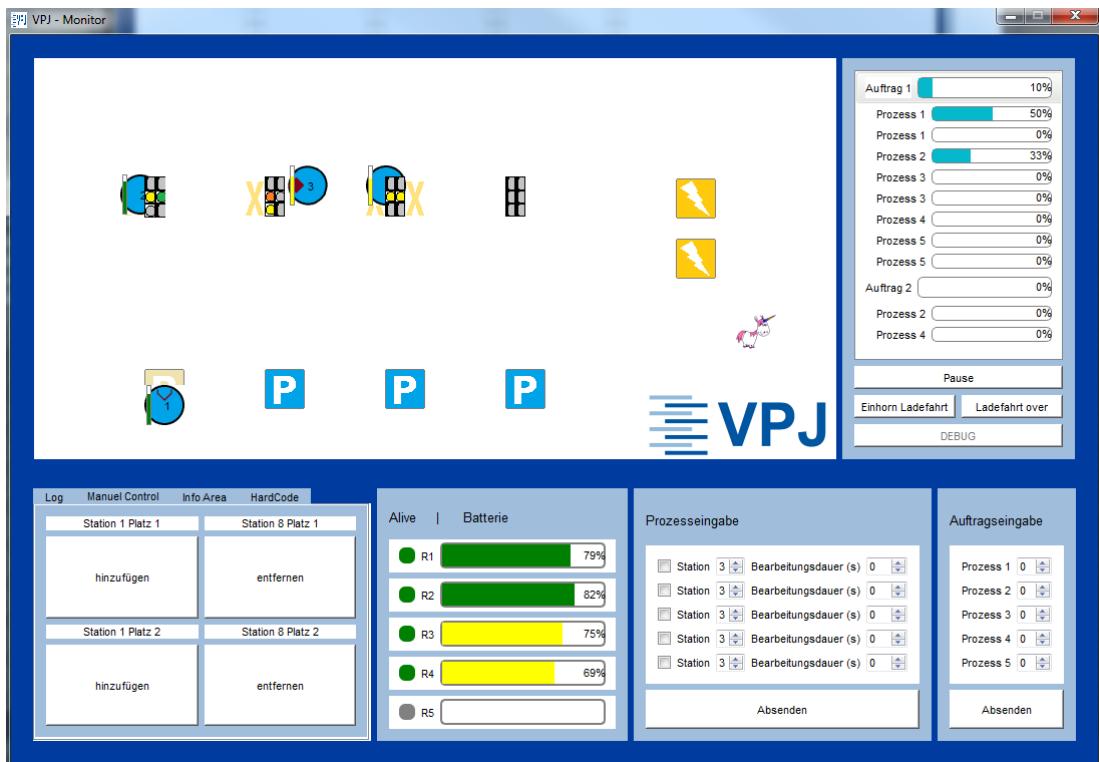


Abbildung 5.3.: Übersicht Visualisierung

und 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.

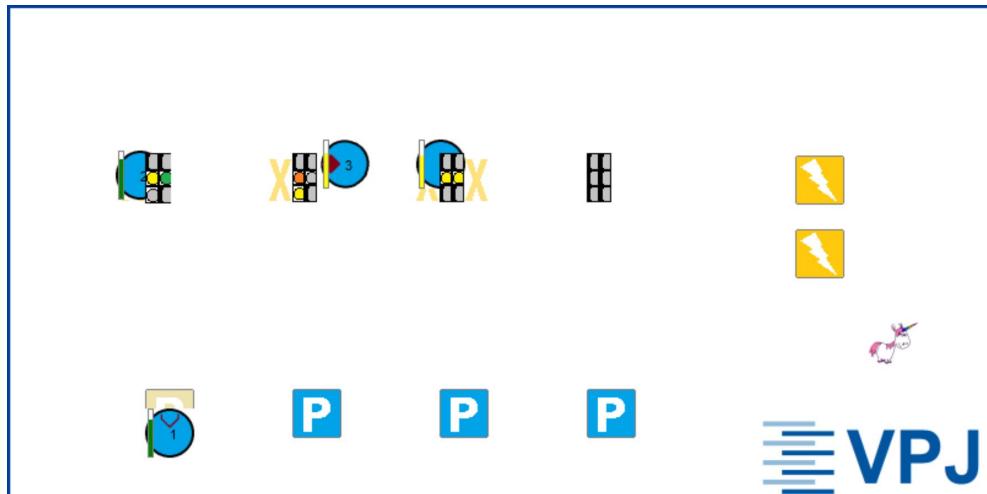


Abbildung 5.4.: Visualisierung Live-View

### 5.3.1. Robotino

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

## 5. 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 5.5 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 orangefarbenem Rand abgebildet, der bei einem anderen Robotino als Hindernis angezeigt wird. Wenn kein Hindernis erkannt wird bleibt der Rand schwarz.

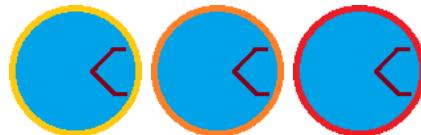


Abbildung 5.5.: Robotino Darstellung bei verschiedenen Hinderniserkennungen

Die Darstellung des Robotinos hängt, neben den Hindernistypen, auch von verschiedenen Stati ab. So wird ein defekter Robotino mit einer Flamme gekennzeichnet (Abb. 5.6 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 5.6 links abgebildet angezeigt. Hierbei wird unterschieden ob der Greifer des Robotinos offen (links) oder geschlossen (rechts) ist.



Abbildung 5.6.: 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 5.1 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 das 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 5.1: Robotino Drehung

```

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

```

## 5. Visualisierung

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

Zusätzlich zur Position beinhaltet jeder Robotino in der Mitte eine Zahl, die die eindeutige RoboterID repräsentiert. Somit kann sofort erkannt werden, welcher Robotino dargestellt ist. Außerdem ist an der linken Seite der Akkustand als vertikaler Balken dargestellt. Der Akkustand ist gleich dem, der unten im Roboterstatus-View angezeigt wird und wird dort näher beleuchtet. Am Akkustand am Robotino kann direkt erkannt werden, welcher Robotino welchen Akkustand besitzt ohne erst die IDs abgleichen zu müssen.

### 5.3.2. Parkplätze und Ladestationen

Im unteren Bereich der Live-View befinden sich die vier Parkplätze für die Robotinos. Je nach Zustand werden diese verschieden dargestellt. Sobald ein Parkplatz für einen Robotino reserviert wird, oder durch einen Robotino belegt ist wird dieser in hellgelb abgebildet. Ein freier Parkplatz wird, wie in Abbildung 5.7 rechts, blau dargestellt.

Eine Aktualisierung erfolgt über den Aufruf der Funktion UpdateParkplatz() in der MainWindow.



Abbildung 5.7.: Visualisierung Parkplatz

Wie die Parkplätze werden auch die Ladestationen entweder reserviert (Abb. 5.8 oben) in blassem hellgelb dargestellt, oder in kräftigem Gelb für freie Ladestationen (Abb. 5.8 unten).



Abbildung 5.8.: Visualisierung Ladestation

### 5.3.3. Stationen

Die Stationen enthalten Informationen über die Werkstücke, die an ihnen bearbeitet werden. Es gibt insgesamt 8 Stationen, wobei immer zwei Stationen horizontal zusammengefasst sind. Ein solcher Stationsverbund ist in Abbildung 5.9 abgebildet. Der oberste Kreis einer Station repräsentiert den RFID-Kopf. Darunter befinden sich die beiden Arbeitsplätze. Jeder Arbeitsplatz enthält in seiner Farbkodierung eine Information über das Werkstück. In der Abbildung sind alle möglichen Farbkodierungen dargestellt.

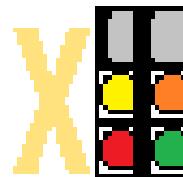


Abbildung 5.9.: Stationen mit verschiedenen Arbeitsplatzstatus

Sobald ein Arbeitsplatz reserviert wird, da sich ein Werkstück auf dem Weg dahin befindet, ist dieser Gelb dargestellt. Ein roter Arbeitsplatz zeigt an, dass dieser defekt ist und nicht mehr angefahren werden kann. Ein defekter Arbeitsplatz wird von der Auftragsplanung nicht mehr berücksichtigt. Diese Einstellung kann mit der Hard-Code Area erzeugt werden. Wenn ein Werkstück an den Arbeitsplatz abgelegt wird befindet sich dieses in Bearbeitung und der Platz wird Orange dargestellt. Sobald die Bearbeitung abgeschlossen ist, wechselt die Farbe des Arbeitsplatz auf Grün und das Werkstück ist bereit zur Abholung. Ein leerer Arbeitsplatz wird in neutralem grau dargestellt.

Über die Funktion `UpdateStationsplatz()` kann die Farbkodierung eines einzelnen Arbeitsplatz anhand des übergebenen Status angepasst werden.

Zusätzlich zu den Arbeitsplatzstatus wird neben jeder Station angezeigt, ob diese reserviert ist. Da nicht zwei Robotinos gleichzeitig zu einer Station fahren sollen, werden diese, sobald sich ein Robotino auf dem Weg dahin befindet oder an einer Station arbeitet, reserviert. Eine reservierte Station ist in der Visualisierung an dem hellgelben X neben der Station zu erkennen. In Abbildung 5.9 ist die linke Station reserviert und die rechte Station frei. Mit der Funktion `UpdateStation()` kann die Reservierung für eine Station angepasst werden.

Im Normalbetrieb, ohne die Zusatzfunktionalität der Hard-Code Area, bewirkt ein Klick auf einen Arbeitsplatz eine Aktualisierung der Timestamp-Area.

## 5.4. Auftragsübersicht

Der rechte Bereich (II) in der Visualisierung enthält die Auftragsübersicht. In Abbildung 5.10 ist zu erkennen, dass sich der Bereich in zwei Teile aufteilt. Der obere Teil ist für die Anzeige der Aufträge und Prozesse zuständig. Im unteren Teil kann über bereitgestellte Buttons interagiert werden.

## 5. Visualisierung

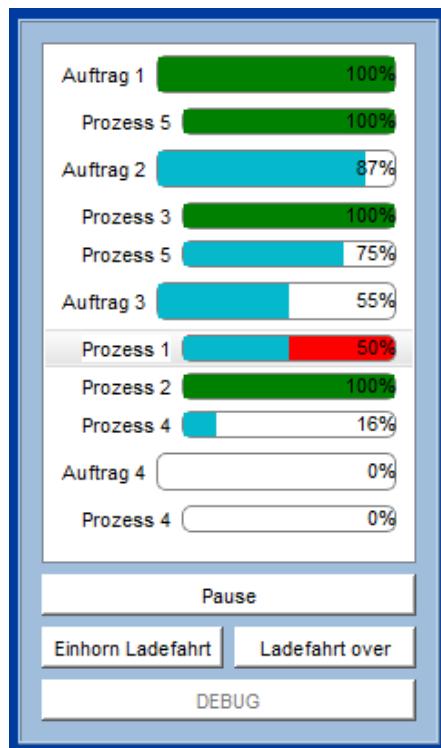


Abbildung 5.10.: Auftragsfortschritt

### 5.4.1. Auftragsdarstellung

In der Abbildung 5.10 ist zu erkennen, dass jeder Auftrag dargestellt wird, inklusive seiner enthaltenen Prozesse. Durch Anklicken eines Auftrags der Liste können die zugehörigen Prozesse eingeblendet (Abb. 5.11 links) oder ausgeblendet (Abb. 5.11 rechts) werden.

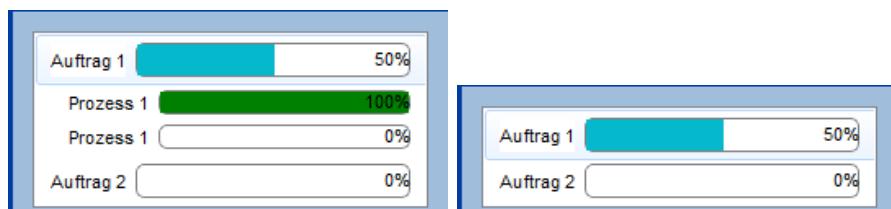


Abbildung 5.11.: Auftrag ein- und ausgeblendet

Der im Auftrag (siehe Kapitel 4.5.3) und Prozess (siehe Kapitel 4.5.2) gespeicherte Fortschritt wird im Fortschrittsbalken aktualisiert, wenn die Funktion UpdateAuftragsFortschritt() innerhalb des Auftragsitem aufgerufen wird. Sobald der Fortschritt 100 Prozent erreicht, wird der Balken grün gefärbt. Der Balken eines geblockten Prozesses wird rot eingefärbt (siehe Abb. 5.10 Auftrag 3 Prozess 1).

### Auftragsitem

Das Auftragsitem (siehe Kapitel 4.7.4) wird in der Auftragsübersicht mit einem Textfeld und nebenstehendem Fortschrittsbalken visualisiert (Abb. 5.12 links).

## 5. Visualisierung

Das Textfeld enthält dabei die Auftragsnummer. Das gesamte Auftragsitem hat eine Höhe von 31 Pixeln, die Breite ist variabel.



Abbildung 5.12.: Auftragsitem (links) und Prozessitem (rechts)

Sowohl Auftrags als auch Prozessitem sind als QListWidgetItem (siehe Kapitel 2.5) implementiert.

Wenn ein QListWidgetItem angeklickt wurde, wird die Funktion `on_AuftragsListWidget_itemClicked()` aufgerufen. In Listing 5.2 ist der Code dargestellt, der dabei für ein Auftragsitem ausgeführt wird. Zunächst wird geprüft, ob das angeklickte Item ein Prozessitem oder ein Auftragsitem ist (Zeile 1). Wenn es sich um ein Auftragsitem handelt, wird die Zeile des angeklickten Items zwischengespeichert (Zeile 3). Anschließend wird für jedes nachfolgende Item geprüft, ob es existiert und ein Prozessitem ist. Solange dabei weitere Prozessitems gefunden werden, werden diese ein- oder ausgeblendet, abhängig davon ob diese zuvor ein- oder ausgeblendet waren. Ansonsten wird die Funktion verlassen.

Listing 5.2: Funktionsblock nach Klicken auf Auftragsitem

```
1 if ((item->whatsThis () == WhatsThis [0] ))
2 {
3     int itemrow = AuftragListWidget->row(item);
4     while (true)
5     {
6         QListWidgetItem* tempitem = AuftragListWidget->
7             item(itemrow+1);
8         if (tempitem == nullptr || (tempitem->whatsThis ()
9             == WhatsThis [0] ))
10        {
11            break ;
12        }
13        tempitem->setHidden (!tempitem->isHidden ());
14        itemrow++;
15        continue ;
16    }
17 }
```

Dadurch können durch Anklicken eines Auftragsitem die zugehörigen Prozessitems angezeigt oder ausgeblendet werden. Prozesse eines anderen Auftrags werden dabei ignoriert.

## Prozessitem

Das Prozessitem (Abb. 5.12 rechts) wird in der Auftragsübersicht ähnlich dem Auftragsitem dargestellt. Der Hauptunterschied ist hierbei, dass das Prozessitem

## 5. Visualisierung

eine Höhe von nur 21 Pixeln besitzt und weiter eingerückt wird, als ein Auftragsitem. Dadurch kann die Hierarchie zwischen Aufträgen und Prozessen einfach erkannt werden.

Wenn nach Anklicken eines QListWidgetItems dieses in der Funktion `on_AuftragsListWidget_itemClicked()` als Prozessitem identifiziert wurde, wird der in Listing 5.3 dargestellte Code ausgeführt.

Listing 5.3: Funktionsblock nach Klicken auf Prozessitem

```
else if (item->whatsThis() == WhatsThis[1])
{
    Prozessitem *pi = dynamic_cast<Prozessitem*>(
        AuftragListWidget->itemWidget(item));
    QPushbutton *p = GetPressedHardCodeButton();
    if (p == ButtonBlockAuftrag)
    {
        pi->prozess->SetBlocked();
        pi->UpdateProzessBlocked();
    }
}
```

Dabei wird das QListWidgetItem zunächst dynamisch in Zeile 3 auf ein Prozessitem gecastet.

In C++ gibt es verschiedene Wege ein Objekt in ein anderes zu casten. Die geläufigsten (regulären) Castvorgänge sind `reinterpret_cast` oder `const_cast`. Neben diesen gibt es Funktionen wie `dynamic_cast` oder `static_cast`. Ein regulärer Cast versucht, aus allen weiteren möglichen regulären Castvarianten eine geeignete zu finden, die funktioniert. `Dynamic_cast`, oder `static_cast` wird hierbei nicht berücksichtigt. Der Unterschied zwischen `static_cast` und `dynamic_cast` ist hauptsächlich, dass für einen `static_cast` bekannt sein muss, in was das Objekt gecastet werden soll. Ein `dynamic_cast` hingegen kann immer ausgeführt werden und liefert einen Nullpointer zurück, wenn der Cast fehlschlägt. In diesem Fall wurde die weniger verbreitete Funktion `dynamic_cast` gewählt, da nur diese alle Anforderungen zur Laufzeit unterstützt (vgl. [Cas17]).

Der Cast ist notwendig, da das allgemeine QListWidgetItem nicht auf die Funktionen zugreifen kann, die das Prozessitem zur Verfügung stellt.

Nachdem sichergestellt wurde, dass das Objekt ein Prozessitem ist, wird in Zeile 4 ausgewertet, ob der Hard-Code Button zum blockieren eines Prozesses aktiv ist und anschließend dieser Prozess dahingehend aktualisiert.

Bei einem Doppelklick auf ein QListWidgetItem wird die Funktion `on_AuftragsListWidget_itemDoubleClicked()` aufgerufen (siehe Listing 5.4). Hierin wird überprüft, ob das angeklickte Objekt ein Prozessitem war und dieses gegebenenfalls gecastet. Auf dem gecasteten Objekt kann dann ein Signal emittiert werden, das eine Aktualisierung der Timestamps bewirkt.

Listing 5.4: Funktionsblock nach Doppelklick auf Prozessitem

## 5. Visualisierung

```
1 Prozessitem *pi = dynamic_cast<Prozessitem*>(  
2     AuftragListWidget->itemWidget(item));  
2 emit GetTimestampsOnProzessClicked(pi->prozess->lfId);
```

### 5.4.2. Buttons der Auftragsübersicht

Um die im Einhorn (Robotino 5) fest einprogrammierte Ladefahrt zu starten, wurden die Buttons „Einhorn Ladefahrt“ und „Ladefahrt over“ implementiert. Jede Betätigung des Buttons „Einhorn Ladefahrt“ bewirkt ein einmaliges Senden eines definierten Auftrags an Robotino 5 (siehe Kapitel 3.2). Dazu wird die Funktion EinhornLaden() im UDP-Handler genutzt.

Über eine Betätigung des Buttons „Pause“ wird die Auftragsvergabe in der State-Machine der Fertigungsplanung pausiert. Somit werden keine Transportaufträge mehr an die Robotinos versendet. Ladefahrten oder eine Parkplatzfahrt werden weiterhin zugelassen. Der Pause Button wird getoggelt, das heißt wenn er betätigt wird, bleibt er bis zur erneuten Betätigung aktiv. Um zu erkennen, dass die Auftragsplanung pausiert ist, wird der Button im aktvierten Zustand, wie in Abbildung 5.13 zu sehen, Rot hervorgehoben.



Abbildung 5.13.: Pause Button gedrückt

Die Auftragsplanung wird beispielsweise pausiert, wenn ein Robotino ausgefallen ist und über die Hard-Code Area die Fertigungsstraße wieder aufgeräumt werden muss.

## 5.5. Tab-View

Der Bereich III unten links wurde als Tab-View umgesetzt. Es kann jederzeit einer aus vier verschiedenen Tabs ausgewählt werden, welcher angezeigt wird. Zwei der Tabs dienen der Anzeige, über die anderen beiden kann das Programm bedient werden.

Auf die einzelnen Tabs wird in den folgenden Abschnitten genauer eingegangen.

### 5.5.1. Log-View

Der erste Tab, der auch bei Programmstart geöffnet ist, beinhaltet das Log (Abbildung 5.14). In diesem Tab ist keine Eingabe möglich. Im scrollbaren Textfenster werden zu Programmalaufzeit Logeinträge dargestellt, die durch den Programmablauf generiert wurden.

## 5. Visualisierung

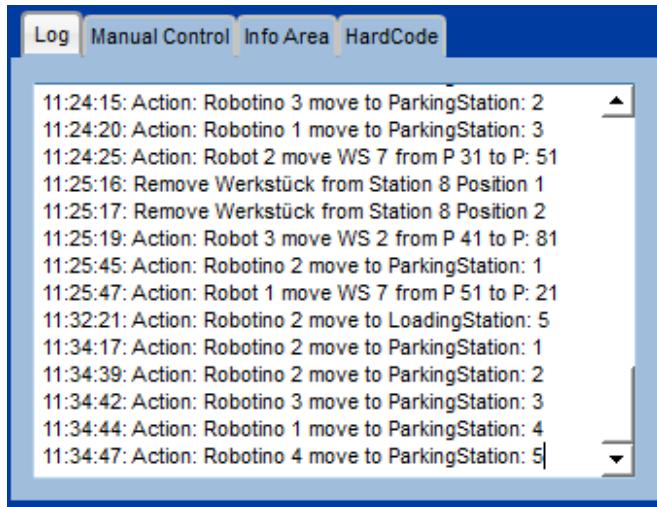


Abbildung 5.14.: Tab: Log View

Über die Funktion WriteToDebugTextArea() kann ein übergebener String an das Textview angehängt werden. So wird im Textfeld eine neue Zeile hinzugefügt. Dem übergebenen String wird zuvor der aktuelle Zeitstempel vorangestellt.

Eine Liste der vorkommenden Logeinträge ist in Tabelle 5.1 zu finden.

Aus der Tabelle kann weiterhin entnommen werden, dass die einzelnen Logeinträge in Kategorien eingeteilt werden können. Die Kategorie Initialisierung zeigt an, dass das Programm erfolgreich gestartet wurde und bereit ist für Eingaben vom Benutzer.

Die Logeinträge der Kategorie Feedback zeigen dem Benutzer, dass eine von ihm getätigte Eingabe erfolgreich interpretiert und ausgeführt wurde. Dazu zählt das Hinzufügen oder Entfernen von Werkstücken an die Stationen oder das Hinzufügen von Prozessen oder Aufträgen.

Die Aktionen beschreiben die von den Robotinos ausgeführten Tätigkeiten und werden immer bei erfolgreicher Auftragsvergabe an einen Robotino dem Log hinzugefügt.

Um eine Warnung zu erzeugen, muss ein Roboter entweder die Kommunikation verloren haben, oder aktiv auf defekt geschaltet worden sein.

Auf einen Error muss individuell reagiert werden. Während des regulären Betriebs sollte keiner der Errors auftreten. Ein Error zeigt somit eine Fehlfunktion des Programms an, und weist auf den Moment des Auftretens hin.

Die Roboter-Error werden vom Robotino gesendet. Auf diese muss der Anwender händisch reagieren. Die einzelnen Errortypen und die nötige Reaktion sind in Kapitel 3.3 beschrieben.

Während des normalen Betriebs besteht das Log zum größten Teil aus Einträgen der Kategorie Aktion, ergänzt um einzelne Feedback-Einträge (siehe Snapshot in Abb. 5.14).

## 5. Visualisierung

Nr	Kategorie	Nachricht
1	Initialisierung	Fertigungsplanung initialisiert
2	Initialisierung	Datenbank gestartet
3	Initialisierung	Fertigungsplanung gestartet
4	Feedback	Prozess hinzugefügt
5	Feedback	Auftrag hinzugefügt
6	Feedback	Add Werkstück to Station 1 Position 1
7	Feedback	Remove Werkstück from Station 8 Position 1
8	Aktion	Action: Robotino 1 move to LoadingStation: 5
9	Aktion	Action: Robotino 1 move to ParkingStation: 1
10	Aktion	Action: Robotino 1 move WS 2 from Position 11 to Position 22
11	Warnung	Warning: Robot 1 died
12	Warnung	Warning: Robotino 1 exploded
13	Info	Info: Robotino 1 repaired
14	Error	ERROR: Could not set Station Place ready after Prozess Timer elapsed
15	Error	ERROR: Could not set Station Place full after Greifer opened
16	Error	ERROR: Could not set Source Station Place free after Robot Greifer closed
17	Error	ERROR: Could not set Loading Station Place free after Robot drove away
18	Error	ERROR: Robot not found
19	Error	ERROR: Could not set Station place free from Hard Code Button
20	Error	ERROR: Could not set Station place defect from Hard Code Button
21	Error	ERROR: Could not set Station free from Hard Code Button
22	Error	ERROR: Could not set Loading Station Place reserved after sending Task to Robot
23	Error	ERROR: Could not set Parking Station Place reserved after sending Task to Robot
24	Error	ERROR: Could not set Target Station Place reserved after sending Task to Robot
25	Error	ERROR: Could not set Source Station Place reserved after sending Task to Robot
26	Roboter Error	ROBOTERROR: Robot 1 lost Werkstueck
27	Roboter Error	ROBOTERROR: Robot 1 was not able to grab Werkstueck
28	Roboter Error	ROBOTERROR: Robot 1 was not able to plan Route

Tabelle 5.1.: Logeinträge

### 5.5.2. Manual Control

Der zweite Tab der Tab-View, Abbildung 5.15, enthält Buttons zum manuellen Hinzufügen und Entfernen von Werkstücken.

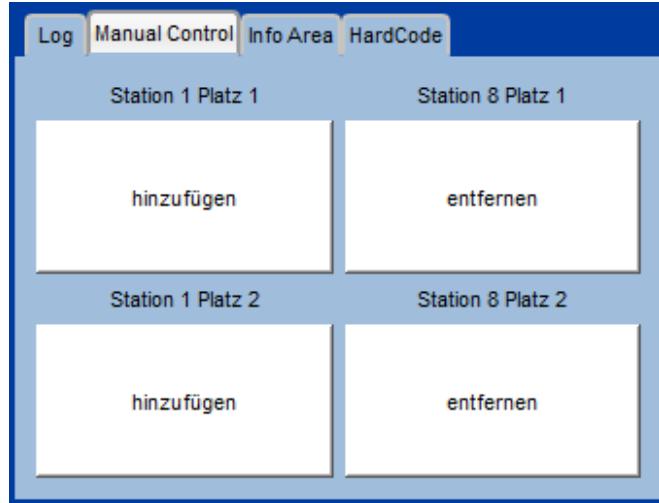


Abbildung 5.15.: Tab: Manual Control

Über die beiden linken Buttons können dem Rohtellager einzelne Werkstücke zugeteilt werden. Dazu muss zuvor ein Werkstück händisch der realen Fertigungsstraße an entsprechender Position eingelegt werden.

Um die fertig bearbeiteten Teile aus Station 8 zu entfernen, können die beiden rechten Buttons verwendet werden.

Über einen Action-Listener wird beim Klick auf einen der Buttons eine zugeordnete Funktion, z.B. `on_S1P2add_clicked()`, aufgerufen. In der aufgerufenen Funktion wird ein Signal emittiert, dass in der Fertigungsplanung ein Werkstück an entsprechender Stelle der Fertigungsstraße hinzufügt oder entfernt.

### 5.5.3. Timestamp-Area

Im dritten Tab der Tab-View erfolgt eine Anzeige der in der Datenbank gespeicherten Timestamps. Es gibt zwei Arten, Timestamps anzeigen zu lassen. Diese sind in Abbildung 5.16 dargestellt.

Es können entweder alle Timestamps eines bestimmten Werkstücks (Abb. 5.16 links) oder alle Timestamps einer Station (Abb. 5.16 rechts) angezeigt werden.

Im Timestamp ist beschrieben, zu welcher Zeit sich ein Werkstück an welcher Station an-, bzw. abgemeldet hat. Man kann feststellen, dass sich an Station 1 nur Werkstücke abmelden und an Station 8 nur Werkstücke anmelden, da diese hier jeweils händisch ergänzt bzw. entnommen werden.

Um die Timestamps einer Station anzuzeigen, muss ein leerer Arbeitsplatz oder der RFID-Platz angeklickt werden. Dadurch wird die Funktion `SetTimestamps()` aufgerufen, welche das Textview aktualisiert.

## 5. Visualisierung

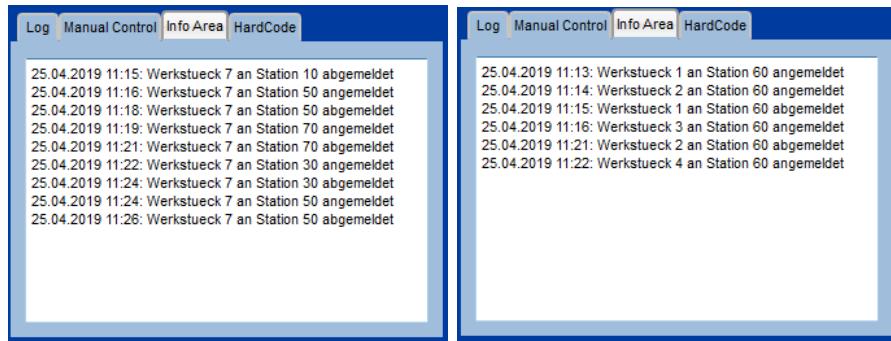


Abbildung 5.16.: Tab: Info Area

Timestamps für ein spezielles Werkstück können angezeigt werden, indem ein dem Werkstück zugeordneter Arbeitsplatz angeklickt wird. Dabei ist es egal, ob sich das Werkstück gerade auf diesem befindet, oder nur eine Reservierung vorliegt. Alternativ kann, um auch abgeschlossene Werkstücke darstellen zu können, auf das zugehörige Prozessitem in der Auftragsübersicht ein Doppelklick gemacht werden. Dadurch werden die zugehörigen Timestamps dargestellt.

### 5.5.4. Hard-Code Bereich

Der Hard-Code Tab wird benötigt, um die Programmlogik gezielt zu umgehen, oder Stati „hart“ zu überschreiben oder zu setzen. Dazu wurden die in Abbildung 5.17 dargestellten Buttons implementiert. Jeder Button hat eine eigene Funktion und wird in den folgenden Abschnitten einzeln beleuchtet.

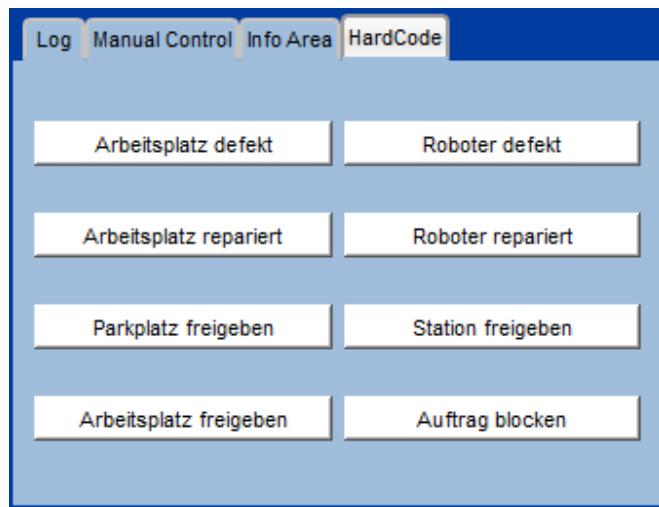


Abbildung 5.17.: Tab: Hard-Code

Die Betätigung eines Buttons versetzt die Visualisierung in den „Hard-Code Modus“. Der betätigte Button bleibt dabei gedrückt und indiziert, welche Funktionalität aktiv ist. Um nicht versehentlich im Hard-Code Modus Dinge zu überschreiben wird das Live-View rot eingefärbt. Ein Vergleich der normalen Visualisierung und des Hard-Code Modus ist in Abbildung 5.18 dargestellt. Es ist gut zu erkennen, dass der Hard-Code Modus nicht übersehen werden kann.

## 5. Visualisierung

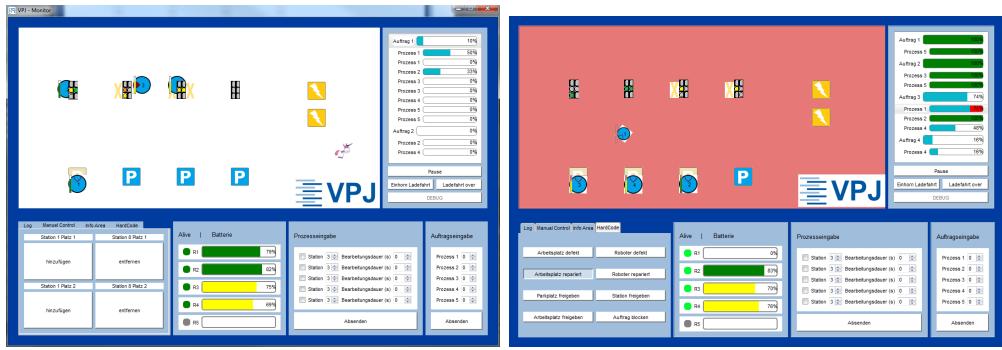


Abbildung 5.18.: Vergleich Visualisierung - Normal VS Hard-Code Modus

Der Hard-Code Modus kann durch Betätigen des aktiven Buttons wieder verlassen werden.

Um sicherzustellen, dass zeitgleich immer nur eine Hard-Code Funktionalität aktiv ist, wird bei Pressen eines Buttons, nach der Button-Listenerfunktion, zuerst die Funktion ResetButtonsExceptOne() aufgerufen, wenn der Button noch nicht aktiv war. In dieser Funktion wird der Hintergrund der Live-View rot eingefärbt. Anschließend werden alle Buttons die sich, neben dem Angeklickten, im Hard-Code Tab befinden deaktiviert.

Die Funktion GetPressedHardCodeButton() liefert für den Fall, dass ein Button aktiviert ist, diesen als Rückgabewert zurück.

### Arbeitsplatz im Hard-Code

Wenn der Button „Arbeitsplatz defekt“ aktiviert, und ein Arbeitsplatz angeklickt wird, so wird der Status dieses Arbeitsplatzes in der Fertigungsstraße zu defekt gesetzt. Die Visualisierung wird dadurch ebenfalls aktualisiert.

Gegenteilig dazu kann, über den Button „Arbeitsplatz repariert“, ein zuvor defekt geschalteter Arbeitsplatz durch Anklicken wieder freigegeben werden.

Um bei Anklicken des Arbeitsplatzes nicht nur den Timestamp auszugeben, wird in der Funktion StationClicked() zunächst über den Rückgabewert der Funktion GetPressedHardCodeButton() ausgewertet, ob einer der beschriebenen Buttons nicht aktiviert ist.

### Robotino im Hard-Code

Ähnlich dem Arbeitsplatz kann über den Button „Roboter defekt“ der Status des angeklickten Robotinos auf defekt gesetzt werden. Dieser wird dadurch von der Auftragsplanung ausgeschlossen und bekommt als Visualisierung die Flamme.

Ein Robotino muss defekt geschaltet werden, wenn sich dieser aufgehängt hat, die Kommunikation zum Robotino fehlschlägt oder andere Probleme an dem Robotino auftreten. Ein Entfernen eines nicht defekt geschalteten Robotinos kann zu einer Auftragsvergabe an den nicht vorhandenen Roboter führen. Durch die

## 5. Visualisierung

implementierten Timeouts würde das Programm zwar weiterlaufen, allerdings in deutlich verlangsamter Form, sobald es dem defekten Robotino einen Auftrag erteilen möchte.

Der Button „Roboter repariert“ aktiviert den angeklickten (defekten) Robotino wieder.

Die dazu verwendete Funktion heißt RobotClicked().

### Freigabefunktionen

Nach einem Ausfall eines Robotinos, während dieser bereits einen Auftrag erhalten hatte, müssen eventuelle Reservierungen gelöscht werden. Dafür sind die Buttons „Parkplatz freigeben“, „Arbeitsplatz freigeben“, und „Station freigeben“ vorgesehen.

Der Button „Station freigeben“ bewirkt, dass die angeklickte Station nicht mehr von einem Robotino blockiert wird. Dazu wird in Datenbank und Fertigungsplanung der Stationsstatus als frei markiert. Um eine Station freizugeben kann ein beliebiger zur Station gehöriger Arbeitsplatz angeklickt werden.

Ebenso wird mit Parkplätzen und Ladestationen verfahren. Da sich diese in der Implementierung nicht unterscheiden, werden beide über den Button „Parkplatz freigeben“ wieder auf frei gesetzt.

Mit dem Button „Arbeitsplatz freigeben“ können reservierte oder belegte Arbeitsplätze durch Anklicken zu freien Arbeitsplätzen zurückgesetzt werden.

### Auftrag blocken

Über den Button „Auftrag blocken“ kann ein Prozess eines Auftrags abgebrochen werden. Dazu muss im Hard-Code Modus der abzubrechende Prozess in der Auftragsübersicht angeklickt werden. Dieser wird dann rot markiert und intern als geblockt behandelt.

Ein Auftrag muss geblockt werden, wenn während des Ablaufs ein unvorhergesehenes Ereignis die weitere Bearbeitung des Auftrags verhindert. Dazu zählt beispielsweise ein Totalausfall des bearbeitenden Robotinos, Verlust des Werkstücks oder anderweitig fehlgeschlagener Transport.

Wenn ein Auftrag geblockt wurde, sollte das zugehörige Werkstück aus der Fertigungsstraße entfernt werden.

## 5.6. Roboterstatus

In Bereich IV wird der Status jedes Robotinos dargestellt. Dabei ist auf der linken Seite der Alive-Status der Robotinos, und auf der rechten Seite der Akkustand dargestellt (Abb. 5.19).

## 5. Visualisierung

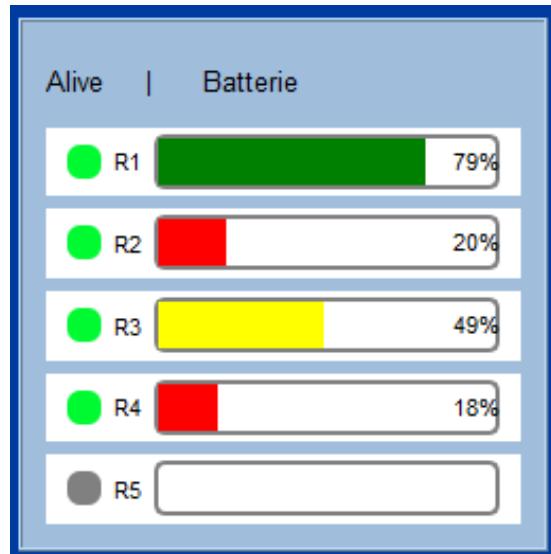


Abbildung 5.19.: Batterie und Statusanzeige

### 5.6.1. Akkustand

Innerhalb der MainWindow wird die Funktion UpdateRobotinoAkku genutzt, um den Akkustand zu aktualisieren. Dabei wird unterschieden, wie hoch der Akkustand ist. Bei einem Akkustand von kleiner als 30 Prozent wird der Fortschrittsbalken rot eingefärbt (Abb. 5.19 R2, R4). Der Robotino wird bei <25 Prozent Akku zur Ladestation geschickt.

Ein Akkustand von über 77 Prozent wird, wie in Abbildung 5.19 R1, grün dargestellt. Bei einem Akkustand zwischen 30 und 77 Prozent wird der Fortschrittsbalken gelb dargestellt (Abb. 5.19 R3).

Die Anpassung der Farbe geschieht über das Setzen des Stylesheet für den Fortschrittsbalken. In Listing 5.5 ist das implementierte Stylesheet für alle Fortschrittsbalken dargestellt.

Listing 5.5: Progressbar Stylesheet

```
QProgressBar {  
    border: 2px solid grey;  
    border-radius: 5px;  
    text-align: right;  
}  
  
QProgressBar::chunk {  
    background-color: #05B8CC;  
    width: 2px;  
}  
  
font: 8pt "OpenSans";  
background-color: rgb(160 ,190 , 220);
```

## 5. Visualisierung

Indem der in Listing 5.6 dargestellte Code in der UpdateRobotinoAkku() Funktion ausgeführt wird, wird die Farbe des jeweiligen Fortschrittsbalken-Chunk überschrieben und somit angepasst. In Zeile 1 erfolgt die Aktualisierung des Fortschrittsbalken in der Roboterstatus-View. Zeile 2 zeigt die anschließende Aktualisierung des vertikalen Fortschrittsbalken am Robotino.

Listing 5.6: Stylesheet Aktualisierung der Akku-Progressbar

```
P->setStyleSheet ("QProgressBar::chunk {background-color :  
    green }");  
2 Pklein->setStyleSheet ("QProgressBar::chunk {background-  
    color: green }");
```

### 5.6.2. Alive-Status

Der linke Bereich des Roboterstatus-View enthält die Information zum Alive-Status des Robotinos (vgl. 4.4). Die dargestellte grüne LED blinkt, während der Roboter als „lebend“ gilt. Das Blinken ist in Abbildung 5.20 abgebildet.



Abbildung 5.20.: Roboterstatus-LED blinkend

Über die Funktion UpdateRobotinoAlive() kann die LED aktiviert oder deaktiviert werden. Wenn der Roboter nicht mehr „lebend“ ist wird eine Meldung an das Log geschrieben und die LED grau gesetzt (Abb. 5.19 R5).

Über den internen blink-Timer wird alle 500 ms die Funktion ToggleAliveRobotino() aufgerufen. In dieser Funktion wird für alle Robotinos, die „lebend“ sind, die LED zwischen hellgrün und dunkelgrün gewechselt. Dies geschieht über eine Anpassung des Stylesheets von der LED (siehe Listing 5.7).

Listing 5.7: Stylesheet Aktualisierung der Alive-LED

```
toggler ? AliveR1->setStyleSheet ("QPushButton {background-  
    color: rgb(0,250,50); border-radius: 6px;}") : AliveR1  
->setStyleSheet ("QPushButton {background-color: green;  
    border-radius: 6px;}");
```

## 5.7. Prozesseingabe

Über den Bereich der Prozesseingabe (V) kann der Fertigungsplanung ein neuer Referenzprozess hinzugefügt werden (siehe Abb. 5.21). Durch die Markierung auf der linken Seite können die Prozessschritte ausgewählt werden, die dem Prozess hinzugefügt werden sollen. Über das UI können Prozesse mit 1 bis 5 Prozessschritten erzeugt werden. Nach Auswahl des Prozessschritts kann eine Station

## 5. Visualisierung

zwischen 2 und 7 ausgewählt werden. Sollten zwei aufeinanderfolgende Prozessschritte die selbe Station beinhalten, wird ein Fehler generiert. Zuletzt kann im rechten Eingabefenster die Bearbeitungsdauer an der Station zwischen 1 und 99 Sekunden ausgewählt werden.

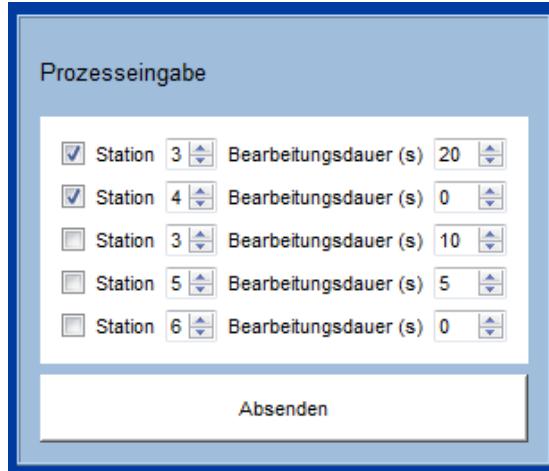


Abbildung 5.21.: Visualisierung - Prozesseingabe

Bei Anklicken des Buttons „Absenden“ wird die Funktion `on_sendProzess_clicked()` aufgerufen. In der Funktion werden die Visualisierungseinträge von oben nach unten überprüft und die Prozessschritte zu einem Prozess erzeugt. Der letzte Prozessschritt zu Station 8 wird automatisch ergänzt. Abschließend wird der neue Prozess an die Fertigungsplanung und darüber an die Datenbank gesendet.

## 5.8. Auftragseingabe

In Abbildung 5.22 ist Bereich VI abgebildet, in dem neue Aufträge erzeugt und dem Programm hinzugefügt werden können.

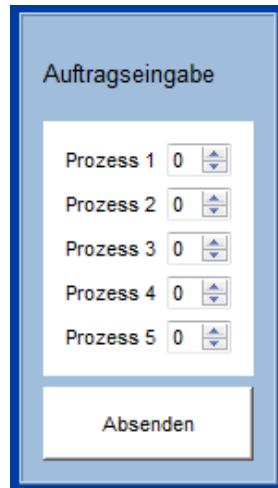


Abbildung 5.22.: Visualisierung - Auftragsvergabe

## 5. Visualisierung

Um einen Auftrag zu erzeugen muss die Anzahl der zu bearbeitenden Referenzprozesse angegeben werden. Über die Tooltips (vgl. Abschnitt sec:tooltips) kann erkannt werden, was für Prozessschritte in den einzelnen Prozessen abgebildet sind.

Bei Betätigung des Buttons „Absenden“ werden die Felder zurückgesetzt und ein neuer Auftrag wird in der Fertigungsplanung erzeugt.

In der Funktion AddAuftragsItem() werden anschließend die UI-Elemente in der Auftragsübersicht erzeugt. Der in Listing 5.8 dargestellte Code enthält die wesentlichen Codezeilen der Funktion.

Listing 5.8: Funktionsinhalt AddAuftragsItem

```
QListWidgetItem *listWidgetItem = new QListWidgetItem(1
    AuftragListWidget);
2 AuftragListWidget->addItem(listWidgetItem);
Auftragsitem *auftragsItem = new Auftragsitem(a);
4 auftragsItem->SetLabelText("Auftrag " + QString::number(a
    ->id));
auftragsItem->UpdateAuftragsFortschritt();
6 listWidgetItem->setSizeHint(auftragsItem->sizeHint());
AuftragListWidget->setItemWidget(listWidgetItem,
    auftragsItem);
8 listWidgetItem->setWhatsThis(WhatsThis[0]);

10 int itemrow = AuftragListWidget->row(listWidgetItem);
foreach (Prozess *p, a->Prozesse)
12 {
    Prozessitem *prozessItem = new Prozessitem(p);
    prozessItem->SetLabelText("Prozess " + QString::number(
        p->referenzId));
    prozessItem->UpdateProzessFortschritt();

16     QListWidgetItem *widgetItem = new QListWidgetItem();
18     AuftragListWidget->insertItem(itemrow+1, widgetItem);
20     AuftragListWidget->setItemWidget(widgetItem,
        prozessItem);
22     widgetItem->setSizeHint(prozessItem->sizeHint());
     widgetItem->setHidden(true);
24     widgetItem->setWhatsThis(WhatsThis[1]);
     QString temp = "Werkstueck: " + QString::number(p->
        lfdId);
     prozessItem->setToolTip(temp);
     itemrow++;
28 }
```

In Zeile 1 wird ein neues QListWidgetItem erzeugt, welches als Elternwidget das

## 5. Visualisierung

AuftragsListWidget gesetzt bekommt. Dieses Item wird nun dem Elternwidget AuftragsListWidget hinzugefügt. In den Zeilen 3 bis 5 wird ein neues Objekt vom Auftragsitem erzeugt, welches dem AuftragsListWidget hinzugefügt wird. Dabei ist wichtig, die Größe des Items explizit zu setzen, da es sonst eine Höhe von 1 Pixel bekommt. In Zeile 7 wird zuletzt das Auftragsitem der Liste angehängt. Das Item bekommt eine eindeutige Kennung, anhand derer ausgewertet wird, dass es sich um ein Auftragsitem handelt.

Nachdem das Auftragsitem erzeugt wurde, werden alle zugehörigen Prozessitems erzeugt. Das Vorgehen ähnelt den Zeilen 1 - 7, nur werden die Prozessitems dem zuvor erzeugtem AuftragsListWidget an einer bestimmten Stelle eingefügt (Zeile 19). Bei der Erzeugung werden die Prozessitems direkt ausgeblendet (Zeile 23). Zuletzt wird der Tooltip (siehe Abschnitt 5.9) für die Prozessitems in Zeile 26 generiert.

Das besondere an dieser Funktion ist, dass dynamisch zur Laufzeit Widget-Elemente erzeugt, und in die in die Visualisierung eingebunden werden.

## 5.9. Tooltips

Um die Benutzerinteraktion zu erleichtern, wurden an zwei Stellen innerhalb der Visualisierung Tooltips implementiert. Ein Tooltip ist ein Hinweistext zu einem Element, der sich beim überfahren mit der Maus (hovern) neben dieser öffnet.

### 5.9.1. Werkstückstooltip

Sobald die Maus über ein Arbeitsplatz gefahren wird, auf dem ein Werkstück liegt, oder eine Reservierung für ein Werkstück existiert, wird über den Tooltip angezeigt, welche ID dieses Werkstück besitzt. In Abbildung 5.23 ist dies anhand einer Reservierung für Werkstück 7 angezeigt.



Abbildung 5.23.: Tooltip eines Werkstücks

Die Tooltips eines Arbeitsplatzes werden über die Funktion UpdateStationToolTip() aktualisiert und über die Fertigungsplanung aufgerufen.

### 5.9.2. Prozessitemtooltip

In Abbildung 5.24 ist exemplarisch der Tooltip für Prozess 4 dargestellt. Dieser wird angezeigt, sobald mit der Maus über den Text des Prozesses gefahren wird.

Der im Tooltip dargestellte Text wird über die Funktion UpdateProzessLabelTooltip() verändert und aktualisiert. Dazu wird jeder Prozessschritt des Prozesses durchgegangen und ein String generiert, der die Station und die Bearbeitungsdauer enthält. Die einzelnen Strings werden untereinander dargestellt.

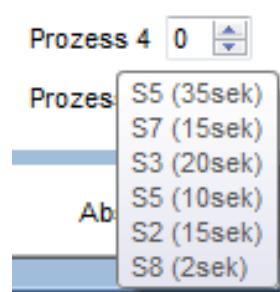


Abbildung 5.24.: Tooltip eines Prozessitem

# 6. Robotersimulation

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, 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 4.3), 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 UDP Handler und Timer für die Zustandswechsel bereitgestellt.

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

## 6.1. Zustandsdiagramm simulierter Roboter

Das Zustandsdiagramm in Abbildung 6.1 beschreibt die Zustände und Transitionen eines simulierten Roboters. Das Diagramm wurde anschließend als State Machine (vgl. 2.4) implementiert.

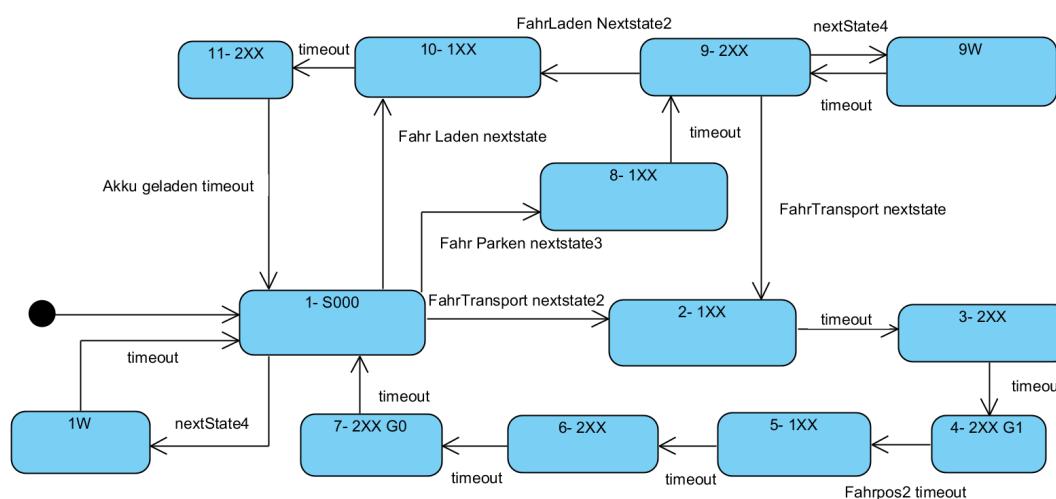


Abbildung 6.1.: Ablaufdiagramm des simulierten Roboters

## 6. Robotersimulation

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 4.6.5 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 definierte 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 der ID 1 in Zustand 2 (Transportauftrag) verlassen werden.

## *6. Robotersimulation*

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.

## 7. Fertigungsrechner

Der Fertigungsrechner ist Teil des Gewerk 1 und besteht hauptsächlich aus der MySQL-Datenbank und der Fertigungsüberwachung. Die MySQL-Datenbank beinhaltet alle relevanten Daten des Fertigungsprozesses. Der Entwurf und die Implementierung der Datenbank wird in Kapitel 8 beschrieben. Über das lokale LAN-Netzwerk kann der Rechner der Fertigungsplanung auf den Fertigungsrechner beziehungsweise die hier abgelegte MySQL-Datenbank zugreifen. Die Fertigungsüberwachung steuert die Ansteuerung und Auswertung der RFID-Schreib-Lese-Köpfe und legt die Daten der Werkstückverfolgung in der Datenbank ab. Mit den Schreib-Lese-Köpfen ist eine Verfolgung der Werkstücke im Fertigungsprozess möglich, da jeder An- und Abmeldevorgang an den Fertigungsstationen 2-7 und dem Rohteil- so wie dem Fertigteilager überwacht und gespeichert wird. Über ein Interface und das Bussystem Profibus-DP sind die Schreib-Lese-Köpfe an den Fertigungsrechner angeschlossen. Die Fertigungsüberwachung wird in Kapitel 9 beschrieben. In der Abbildung 7.1 ist eine Übersicht über die Komponenten des Fertigungsrechners und die Schnittstellen dargestellt.

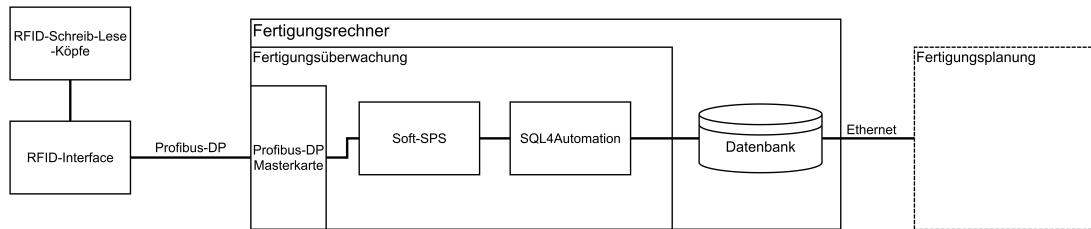


Abbildung 7.1.: Fertigungsrechner und Schnittstellen

# 8. Datenbank

## 8.1. Aufgabe der Datenbank

Die Datenbank soll den aktuellen Zustand der Produktionsanlage abbilden. Dazu gehören die Betriebsmittel, wie zum Beispiel die Fertigungs-Stationen oder die Roboter mit ihren aktuellen Status und der jeweiligen Ladung der Akkumulatoren. Es sollen auch die Verwaltungsdaten, wie Aufträge und Produktionsprozesse in der Datenbank gespeichert werden. Neben dem aktuellen Zustand der Produktionsanlage sollen auch die Fertigungsabläufe dort gespeichert werden. Dies beinhaltet in erster Linie eine Verfolgung der Werkstück durch Timestamps für das An- und Abmelden der Werkstück an den Bearbeitungs-Stationen. Es wird ebenso der Fortschritt, den ein Werkstück im Fertigungsprozess und ein Auftrag insgesamt gemacht hat, gespeichert. Eine weitere Aufgabe der Datenbank ist der Austausch von Daten zwischen dem Fertigungsrechner und der Fertigungsplanung. Alle Daten die zwischen diesen beiden Rechnern ausgetauscht werden, werden in die Datenbank geschrieben.

## 8.2. Konzept

Aus der realen Fertigungsanlage wird ein konzeptionelles Modell als Entität-Relationship-Diagramm modelliert. Das so modellierte ER-Diagramm ist in Abbildung 8.1 zu sehen. Dazu werden insbesondere die Aufgaben aus Abschnitt 8.1 berücksichtigt. Zur Qualitätssicherung wird sich an den drei gängigen Normalisierungsformen für Datenbanken orientiert. Anschließend wird das konzeptionelle Modell in ein logisches Datenschema überführt. Das Datenschema soll den Ansprüchen eines relationalen Datenbankmodells entsprechen. Zur Erstellung des Datenschemas wird die MySQL-Workbenche genutzt und deren Möglichkeit ein relationales Datenbankmodell grafisch zu entwerfen.

## 8.3. Konzeptionelles Modell

Zur Erstellung des konzeptionellen Modells wird für jede Betriebsmittelklasse der realen Anlage ein Entitätstyp angelegt. Es werden folgende Entitätstypen zur Präsentation von Betriebsmitteln angelegt: Roboter, Arbeitsplatz, Parkplatz und Werkstück. Die vier Entitätstypen enthalten die zu speichernden Attribute der Betriebsmittel. So sollen für das Betriebsmittel Roboter zum Beispiel die Attribute „Akkuleistung“ und „Status“ gespeichert werden. Neben den Entitätstypen für

## 8. Datenbank

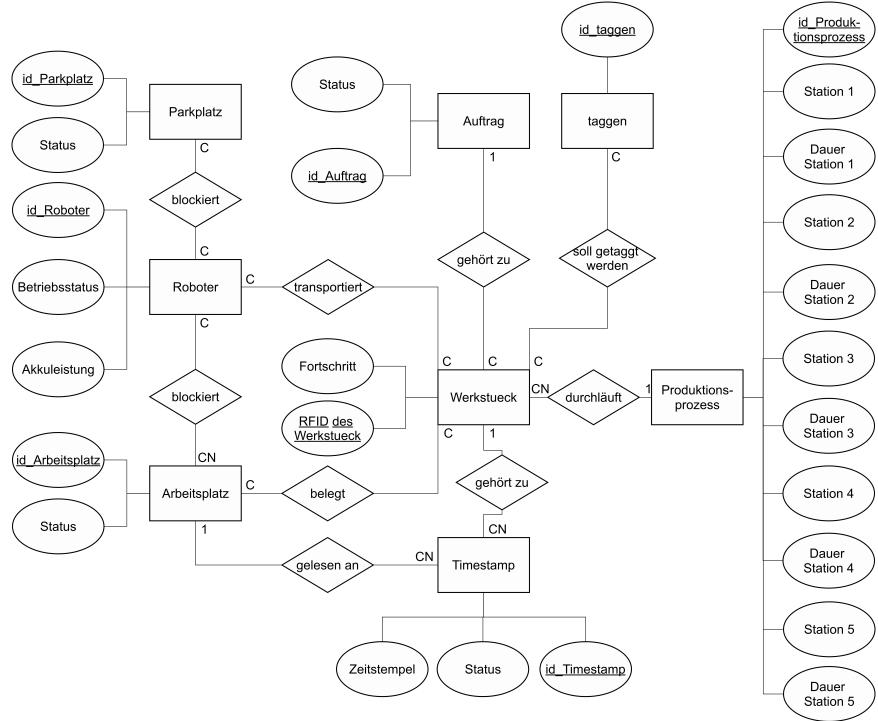


Abbildung 8.1.: Entität-Relationship-Diagramm

die realen Betriebsmittel werden auch Entitätstypen für die Verwaltungsdaten, die zur Fertigungsplanung benötigt werden, angelegt. Hierzu zählen die Entitätstypen: Auftrag und Produktionsprozess. Der Entitätstyp Timestamp dient zur Speicherung der Fertigungsabläufe. Der Entitätstyp taggen dient ausschließlich zur Kommunikation zwischen der Fertigungsplanung und dem Programm für die Ansteuerung der RFID-Schreib-Lese-Köpfe.

Zur Identifizierung der einzelnen Entitäten enthält jeder Entitätstyp mindestens ein Attribut oder eine Kombination von Attributen, die die Entitäten eindeutig unterscheiden. Viele der hier beschriebenen Entitätstypen haben ein aus der realen Welt abgeleitetes Attribut, wodurch die Entitäten unterschieden werden. So ist zum Beispiel die Roboter ID ein Wert der sich aus der Nummerierung der Roboter in der realen Welt ableitet. Es gibt allerdings auch Entitätstypen die eigentlich kein einzelnes Attribut haben, dass sie eindeutig unterscheidet. Als Beispiel hierfür kann der Timestamp genannt werden, wo sowohl der Zeitstempel als auch der Status einen Entität nicht eindeutig identifizieren. Erst in Kombination mit der Beziehung zu einem Werkstück kann der Zeitstempel die Entitäten eindeutig unterscheiden. Zur einfacheren Handhabung des Timestamps wird dem Entitätstyp ein zusätzliches Attribut „id\_Timestamp“ hinzugefügt. Dabei handelt es sich um eine laufende Nummer die keinen zusätzlichen Informationen speichert sondern nur der einfacheren Verwaltung dient. Die Attribute oder Kombinationen von Attributen die die Entitäten eines Entitätstyps eindeutig unterscheiden werden Schlüssel genannt. Wie schon beim Timestamp beschrieben, wurde darauf geachtet das jeder Entitätstyp nur ein Attribut als Schlüssel hat.

Da die Betriebsmittel der realen Welt, ebenso wie die Daten der Fertigungsplanung oder die Fertigungsabläufe, in Beziehung zu einander stehen, werden die

## 8. Datenbank

Entitätstypen mit Beziehungstypen verbunden und so die jeweilige Beziehung deutlich gemacht. Es gibt je nach Art der Beziehung verschiedene Beziehungstypen. Es werden Beziehungstyp-Richtungen mit Kardinalität 1 oder Kardinalität N verwendet. Wobei die Beziehungstyp-Richtungen mit der Kardinalität 1 sowohl optional als auch nicht-optional vorkommen. Alle vorkommenden Beziehungstypen sind in Tabelle 8.1 aufgeführt. Die dort aufgeführten Beziehungen gelten natürlich umgekehrt genauso. Hat Auftrag zu Werkstueck eine 1:CN Beziehung, so hat Werkstueck zu Auftrag eine CN:1 Beziehung. Für die Beziehung zwischen einem Timestamp und einem Werkstueck bedeutet das zum Beispiel, dass ein Timestamp immer zu genau einem Werkstueck gehört, das Werkstueck aber keinen, einen oder mehrere Timestamps haben kann. Es sich also um einen 1:CN Beziehungstyp handelt. Ein Parkplatz hingegen kann immer nur für einen oder keinen Roboter reserviert sein. Genauso kann ein Roboter immer nur an einem oder an keinem Parkplatz sein. Es handelt sich also um eine C:C Beziehung.

Tabelle 8.1.: Beziehungstypen

Beziehungstyp	zwischen
C:C	taggen zu werkstueck roboter zu parkplatz roboter zu werkstueck
C1:CN	arbeitsplatz zu roboter arbeitsplatz zu werkstueck
1:CN	werkstueck zu timestamp arbeitsplatz zu timestamp auftrag zu werkstueck produktionsprozess zu werkstueck

Beim entwerfen des Datenmodells mit einem Entity-Relationship-Diagramm, gilt es verschiedene Punkte zu beachten. Besonders wichtig ist es, Redundanz bei Attributen zu vermeiden. Bei Redundanz treten verschiedene Probleme auf. So müssen die selben Daten mehrfach in die Datenbank geschrieben werden und es wird außerdem unnötig Speicherplatz belegt. Diese Probleme sind aufgrund der kleinen Datenmenge und der geringen Anzahl an Usern bei der hier beschriebenen Produktionsanlage wenig relevant. Redundanz ist auch deshalb zu vermeiden, weil sich ein Problem ergibt, wenn inkonsistente Daten entstehen, weil nicht an allen Stellen, die mehrfach vorhandenen Daten geändert wurden. Um Redundanzen in der Datenbank zu vermeiden, wird das Datenbankmodell normalisiert. Das Normalisieren ist im Abschnitt 8.3.1 beschrieben. Ebenso ist darauf zu achten keine, redundanten Beziehungstypen zu modellieren. Eine Redundanz kann immer dann vermutet werden, wenn zwei Wege von einem Entitätstyp zu einem anderen Entitätstyp möglich sind. Beispielhaft sei hier der Entitätstyp Timestamp genannt, der sowohl eine direkte Beziehung zum Entitätstyp Werkstueck als auch eine indirekte Beziehung über den Entitätstyp Arbeitsplatz hat. Da es sich bei den beiden Beziehungen des Timestamps allerdings um zeitlich abgeschlossene Vorgänge handelt und die Beziehung zwischen Arbeitsplatz und Werkstück den aktuellen Zustand speichert, kann von der Beziehung des Timestamps zum Arbeitsplatz nicht auf das dazugehörige Werkstück geschlossen werden, so dass die direkte Beziehung zum Werkstück notwendig ist.[Jar16]

### 8.3.1. Normalisierung

Anhand von Normalisierungsstufen kann die Qualität eines Datenmodells beurteilt werden. Zur Qualitätssicherung wurden die ersten 3 Normalisierungsstufen auf das in Abschnitt 8.2 beschriebene und in Abbildung 8.1 dargestellte konzeptionelle Datenmodell angewandt. Durch die Anwendung der 3 Normalisierungsstufen entsteht ein Datenmodell, dass in der 3. Normalform vorliegt. Das Vorliegen in der 3. Normalform bietet auch bei der Überführung in das relationale Datenbank-Modell Vorteile Beziehungsweise ist bei Stufe 1 sogar Voraussetzung.

#### 1. Normalform

Die erste Normalform besagt, dass eine Entität keine Attribute besitzen darf, die zur gleichen Zeit mehrere Werte annehmen können. So würde es zum Beispiel gegen die 1. Normalform verstossen, wenn die Zeitstempel als Attribut des Werkstücks angelegt worden wären, da dieses Attribut, je nach dem wie viele Stationen das Werkstück schon durchlaufen hat, mehrere Werte annehmen könnte. Um die 1. Normalform zu erreichen, werden Attribute, die mehrere Werte annehmen können, aus dem Entitätstyp herausgelöst und bilden einen eigenen Entitätstyp, der durch einen Beziehungstyp mit dem ursprünglichen Entitätstyp verbunden ist. Dabei hat die Beziehungstyp-Richtung „Ursprüngliche Entitätstyp zu neuem Entitätstyp“ die Kardinalität N oder CN. Angewandt auf das Beispiel des Zeitstempels wurde, wie in Abbildung 8.1 zu sehen ist, der Zeitstempel in den neuen Entitätstyp Timestamp ausgelagert. Der Beziehungstyp-Richtung von Werkstück zu Timestamp ist CN. Ein Wert, welcher nicht aus einer Liste besteht oder auf andere Weise aus mehreren Werten zusammengesetzt ist, wird als atomarer Wert bezeichnet. Für ein relationales Datenbank-Modell gilt: Jeder Wert eines Attributes muss ein atomarer Wert sein. [Jar16]

#### 2. Normalform

Damit ein Datenmodell in der 2. Normalform vorliegt, muss es sich in der ersten Normalform befinden. Zusätzlich darf jedes Attribut ausschließlich vom Gesamtschlüssel eines Entitätstyps funktional abhängig sein. Funktionale Abhängigkeit bedeutet, dass aus dem Wert von Attribut A sich automatisch der Wert von Attribut B ergibt. Zum Verdeutlichen der zweiten Normalform nutze ich ein etwas konstruiertes Beispiel: Angenommen die Produktionsanlage fährt mit zwei verschiedenen Typen von Robotern. Mit jeweils zwei alten Festo-Robotern und zwei neuen von Studenten entwickelten Robotern. Die ID der Roboter setzt sich nun aus zwei Ziffern zusammen, die jeweils ein eigenes Attribut bilden und zusammen den Gesamtschlüssel für den Entitätstyp Roboter bilden. Die erste Ziffer wäre dabei entweder eine 1 für die Festo-Roboter oder eine 2 für die neuen Roboter. Die zweite Ziffer ist eine laufende Nummer. Wird jetzt zusätzlich das Baujahr gespeichert und sind die Roboter eines Typs jeweils im gleichen Jahr gebaut, so liegt eine funktionale Abhängigkeit zwischen dem Attribut Baujahr und dem Teilschlüssel erste Ziffer vor. Um die zweite Normalform nun herzustellen, wird der

## *8. Datenbank*

Teilschlüssel und das abhängige Attribut herausgelöst und sie bilden einen neuen Entitätstyp. Die zweite Normalform verringert Redundanzen innerhalb des Datenmodells.[HSS01]

### **3. Normalform**

Damit ein Datenmodell in der 3. Normalform vorliegt muss es sich in der 2. Normalform befinden und zusätzlich darf eine Attribut eines Entitätstyps nicht transitiv funktional abhängig sein. So wäre es zum Beispiel möglich, die Attribute des Entitätstyps Produktionsprozesses mit im Entitätstyps Werkstück zu speichern. Es läge dann allerdings eine transitive funktionale Abhängigkeit zwischen den Werten der Attribute Station 1 bis 5 und Dauer Station 1 bis 5 und dem Schlüssel RFID Werkstück vor, da die Attribute Station 1 bis 5 und Dauer Station 1 bis 5 funktional von der id\_Produktionsprozess abhängig wären. Die id\_Produktionsprozess wiederum wäre funktional abhängig vom Schlüssel RFID Werkstück. Deshalb wird dieser Teil herausgelöst und bildet einen eigenen Entitätstyp. Die dritte Normalform verringert die Gefahr von Redundanzen im Datenmodell.[HSS01]

## **8.4. Relationales Datenbankmodell**

Bei einer MySQL-Datenbank handelt es sich um eine relationale Datenbank. Dabei entspricht eine Entität aus dem ER-Diagramm einem Tupel von Werten und der Entitätstyp einem Relationstyp.

So lässt sich nun die Mengen aller Entitäten eines Entitätstyps als Tabelle darstellen. Dabei entsprechen die Überschriften der einzelnen Spalten den Namen der Attribute. Was im ER-Diagramm Attribute genannt wurde, wird nun Eigenschaften genannt. Eine Zeile repräsentiert dann jeweils eine Entität Beziehungsweise ein Tupel von Werten. Wie schon beim ER-Diagramm hat jeder Relationstyp ein oder mehrere Attribute zur eindeutigen Identifizierung der jeweiligen Tupel, die im relationalen Datenbankmodell als Primärschlüssel bezeichnet werden und dem Schlüssel im ER-Diagramm entsprechen. Beziehungstypen werden über sogenannte Fremdschlüsselelemente realisiert. Fremdschlüsselelemente stellen eine zusätzliche Spalte in einer Tabelle dar. Ist in einer Tabelle ein Fremdschlüssel eingetragen, referenziert dieser auf eine Zeile einer anderen Tabelle und stellt so die Beziehung her. Dabei muss stets die referentielle Integrität dieser Verweise gewährleistet sein, dass bedeutet das ein Fremdschlüssel immer auf eine vorhandene Zeile einer anderen Tabelle referenzieren muss oder mit einem NULL-Marker belegt sein muss.

## **8.5. Zugriff auf die Datenbank**

Die MySQL-Datenbank ist auf einem Server, der auf dem Fertigungsrechner läuft, implementiert. Es gibt zwei User die Zugriff auf die Datenbank benötigen. Das

## 8. Datenbank

ist zum einen die Fertigungsüberwachung, die das Auslesen der RFID-Tags übernimmt und die Fertigungsplanung. Die Fertigungsüberwachung befindet sich auf dem selben Rechner wie der Server der Datenbank. Die Fertigungsplanung ist auf einem anderen Rechner realisiert und über Ethernet mit dem Fertigungsrechner verbunden. Die Fertigungsplanung ist mit Qt ausgeführt und hat über eine eingebundene Bibliothek über Ethernet direkten Zugriff auf die Datenbank.

Die Fertigungsüberwachung läuft auf einer Soft-SPS auf dem Fertigungsrechner und ist mit CODESYS programmiert. Ein direkter Zugriff von der Soft-SPS auf die Datenbank ist nicht möglich. Ein Zugriff könnte über einen OPC-Server erfolgen. OPC (Open Platform Communications) ist eine standardisierte Softwareschnittstelle. Sie ermöglicht den Datenaustausch zwischen Steuerungen verschiedener Hersteller aus der Automatisierungstechnik. Hier wurde allerdings ein anderes Verfahren, speziell für den Zugriff auf Datenbanken von Steuerungen aus der Automatisierungstechnik, angewandt. Es handelt sich dabei um den SQL4Automation Connector der Firma Inasoft Systems GmbH. Dieser wurde speziell für den Zugriff von Speicherprogrammierbaren Steuerungen auf eine Datenbank entwickelt und ermöglicht sowohl Lese- als auch Schreibbefehle in SQL direkt auf der SPS zu programmieren. Für die verwendete Steuerung ist schon eine fertige Bibliothek vorhanden, was die Anwendung deutlich erleichtert. In Abbildung 8.2 ist schematisch dargestellt, wie auf die Datenbank zugegriffen wird.

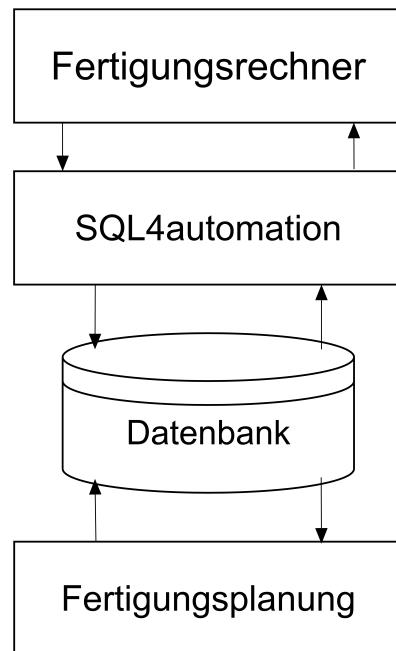


Abbildung 8.2.: Schematische Darstellung des Datenbankzugriffs

## 8.6. Umsetzung des entwickelten Modells

Zunächst wurde anhand des in Abschnitt 8.2 beschriebenen Konzeptes das ER-Diagramm der Datenbank grafisch entwickelt. Das so entstandene ER-Diagramm,

## 8. Datenbank

ist in Abbildung 8.1 zu sehen. Die Entwicklung des ER-Diagramms erfolgte in enger Absprache mit der Fertigungsplanung, da diese einer von zwei Nutzern der Datenbank ist.

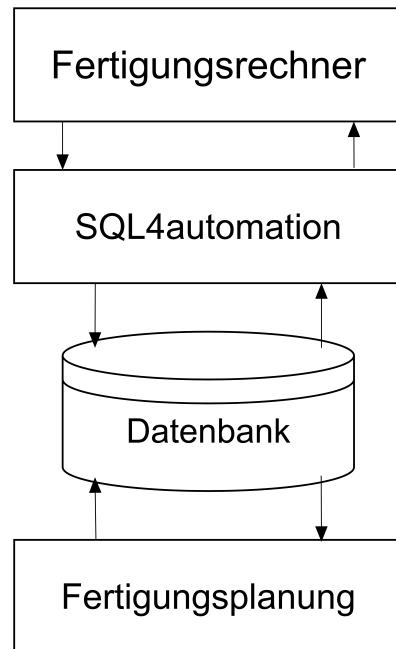


Abbildung 8.3.: Grafisches Modell der Datenbank

Zum Erstellen der Datenbank wurde das Tool MySQL-Workbenche vom Entwickler Oracle Corporation genutzt. Dieses Tool ermöglicht die visuelle Entwicklung, Erstellung und Bearbeitung einer Datenbank. Es können außerdem MySQL-Befehle, zum Beispiel zum Testen, ausgeführt werden. Die visuelle Entwicklung erfolgt über die grafische Eingabe des aus dem ER-Diagramm entwickelten relationalen Datenbankmodells wie in Abbildung 8.3 dargestellt. In der Tabelle 8.2 sind die Symbole, die durch die MySQL-Workbenche verwendet werden, beschrieben. Es können dort die Entitätstypen, welche in das relationale Datenbankmodell transformiert einer Tabelle entsprechen, angelegt werden. Jeder Tabelle können ein Primärschlüssel und Eigenschaften zugewiesen werden. Sowohl dem Primärschlüssel, als auch den Eigenschaften muss ein Datentyp zugewiesen werden. Es können auch weitere Festlegungen für die Eigenschaften getroffen werden. So kann eine Eigenschaft als „NOT NULL“ festgelegt werden, was bedeutet, dass beim Anlegen einer neuen Entität beziehungsweise einer neuen Zeile in der Tabelle diese Eigenschaft immer gesetzt werden muss und auch nicht später gelöscht werden kann. So muss zum Beispiel ein Parkplatz immer einen Status haben. Beim Produktionsprozess, der nicht immer die selbe Länge hat, ist es nicht notwendig dass alle Eigenschaften immer einen Wert haben. So braucht ein Produktionsprozess mit nur 4 Stationen die Eigenschaften „station\_5“ und „dauer\_station\_5“ nicht. Der Primärschlüssel muss selbstverständlich immer vergeben werden und darf nicht mit einem NULL-Marker belegt sein. Auch Beziehungstypen können in der MySQL-Workbenche abgebildet werden. Wird ein Beziehungstyp angelegt, so erzeugt dieser automatisch den, wie in Abschnitt 8.4 beschrieben, dazugehörigen Fremdschlüssel in den jeweiligen Tabellen. Je nach Beziehungstyp dürfen diese

## 8. Datenbank

Tabelle 8.2.: Zeichenelegende MySQL-Workbench Modell

Zeichen	Bedeutung
	Primärschlüssel
	Eigenschaft (darf nicht NULL sein)
	Eigenschaft
	Fremdschlüssel (darf nicht NULL sein)
	Fremdschlüssel
	1:N Beziehung
	1:1 Beziehung

durch einen NULL-Marker belegt sein oder müssen eine Wert haben. So kann bei einer optionalen C1:CN Beziehung, wie sie zum Beispiel von Arbeitsplatz zu Roboter besteht, das Feld für den Fremdschlüssel auch „NULL“ sein. Bei einer 1:CN Beziehung, wie vom Auftrag zum Werkstück, darf der Fremdschlüssel „auftrag\_id\_auftrag“ beim Werkstück nicht mit einem NULL-Marker belegt sein, da jedes Werkstück einem Auftrag zugeordnet werden muss.

Nach dem grafischen Entwurf der Datenbank kann mittels der Option „Forward Engineer to Database“ aus dem grafischen Modell automatisch der entsprechende Code für die Datenbank erstellt werden und die Datenbank ohne weitere Programmierung implementiert werden. Auch nachträgliche Änderungen oder Erweiterungen an der Datenbank können direkt im grafischen Modell erfolgen und über die „Forward Engineer to Database“ Option direkt eingepflegt werden. So bleibt das grafische Modell und die Datenbank immer auf dem selben Stand. Es ist keine zusätzliche Pflege der Dokumentation während des Entwicklungsprozess nötig.

Nach dem Erstellen der Datenbank lässt sich die Struktur der Datenbank im Navigator der MySQL-Workbenche betrachten. Zum Befüllen der Datenbank lassen sich über das Abfragefenster MySQL-Befehle, wie zum Beispiel „INSERT INTO“, ausführen. Beim Befüllen ist insbesondere darauf zu achten, die Tabellen in der richtigen Reihenfolge zu befüllen. So kann ein Werkstück zum Beispiel erst angelegt werden, wenn der dazugehörige Auftrag bereits erstellt wurde.

Über das Abfragefenster lassen sich auch die MySQL-Befehle, die später im CODESYS-Programm auf der Soft-SPS sowie im mit Qt entwickelten Programm auf dem Fertigungsrechner implementiert werden, testen. Auf die MySQL-Befehle die für Qt entwickelt wurden wird in 8.7 eingegangen.

Das Erstellen der MySQL-Befehle lief nach und nach parallel zur Entwicklung der beiden User-Programme der Datenbank. Zunächst wurde die Anforderung an

## 8. Datenbank

den Befehl festgelegt. Anschließend wurde der Befehl erstellt und mit der MySQL-Workbenche getestet. Nach dem erfolgreichen Test wurde der Befehl in seine Zielumgebung, also CODESYS oder Qt, portiert und dort erneut getestet. Beim Testen der Befehle ist insbesondere der Error-Code interessant. Er liefert wichtige Hinweise warum ein Befehl nicht funktioniert. Der Error-Code kann anhand von Error-Code Listen dekodiert werden. Für die Tests in der MySQL-Workbenche übernimmt das dekodieren die Entwicklungsumgebung.

## 8.7. MySQL-Befehle für die Fertigungsplanung

Im folgenden Abschnitt wird zunächst allgemein beschrieben wie die in Qt verwendeten MySQL-Befehle aufgebaut sind und anschließend jeweils ein Befehl der drei Klassen Getter, Setter und Update näher beschrieben. Hier wird nur auf den Aufbau der MySQL-Befehle eingegangen. Wie der Aufruf der Funktion aus Qt heraus funktioniert ist in Abschnitt 4.2 beschrieben.

Es hat sich für die Implementierung in Qt herausgestellt, dass es sinnvoll ist zunächst mit einem SET-Befehl eine Variable zu deklarieren und ihr einen Wert zuzuweisen. Das erhöht die Lesbarkeit des Codes, da die eigentliche Abfrage nicht zerschnitten werden muss.

### 8.7.1. Getter

Im Folgenden wird beispielhaft ein MySQL-Befehl aus einem Getter in Qt beschrieben: Ein Getter fragt Werte aus der Datenbank ab, ohne diese zu verändern. Im Listing 8.1 ist der Code für die Abfrage, ob und welcher Arbeitsplatz an einer Station frei ist, dargestellt. Der Befehl befindet sich in der Funktion GetZielStationsplatzFree(int Station). Die Zuweisung zur Variable „@Station1“ entspricht dabei dem der Funktion übergebenen Wert mit 10 multipliziert, da die einzelnen Plätze wie in Abbildung 3.2 dargestellt kodiert sind. Diese Abfrage liefert eine 1 zurück wenn beide (Zeile 4) oder der erste (Zeile 5) Arbeitsplatz der entsprechenden Station frei sein. Ist nur der zweite Arbeitsplatz frei liefert die Abfrage eine 2 (Zeile 6) und sollten alle Arbeitsplätze belegt sein, wird eine 0 zurückgeliefert (Zeile 7).

Listing 8.1: MySQL-Befehl: Freier Arbeitsplatz

```
1  SET @Station1 = station*10;
2  SELECT
3      (CASE
4          WHEN (SUM(id_Arbeitsplatz))=@Station1+1+@Station1+2 THEN 1
5          WHEN (SUM(id_Arbeitsplatz))=@Station1+1 THEN 1
6          WHEN (SUM(id_Arbeitsplatz))=@Station1+2 THEN 2
7          ELSE 0
8      END)
9  FROM
10     vpj.arbeitsplatz
11 WHERE (id_Arbeitsplatz IN (@Station1+1,@Station1+2)
12         AND Werkstueck_RFID_Werkstueck is NULL);
```

## 8. Datenbank

Bei den MySQL-Gettern handelt es sich meistens um Abfragen die nur einen Wert zurück liefern und keine ganzen Tabellen, so dass die Rückgabewerte direkt eine Aussage treffen und nicht die Tabellen erneut durchsucht werden müssen. Ziel war es, die Abfragen so präzise wie möglich zu gestalten, um die Nachbearbeitung der Abfrage-Ergebnisse weitestgehend zu verhindern. Eine Ausnahme stellen hierbei die Daten da die später als Listen in der Visualisierung ausgegeben werden. Beispielhaft seien hier die RFID-Timestamps mit den dazugehörigen Zeiten und Arbeitsplätzen genannt, die als Liste zurückgegeben werden.

### 8.7.2. Update

Im Folgenden wird beispielhaft ein MySQL-Befehl aus einer Update-Funktion in Qt beschrieben: Eine Update-Funktion verändert einen vorhandenen Eintrag in der Datenbank. Im Listing 8.2 ist der MySQL-Befehl der Funktion UpdateParkplatz(int ID, int status, int roboterid) dargestellt. Mit dieser Funktion kann einem Parkplatz in der Datenbank ein neuer Roboter sowie ein neuer Status zugewiesen werden. Wie auch beim Getter werden die Variablen zunächst mit einem Set-Befehl gesetzt. Wenn ein Roboter einen Parkplatz nicht mehr belegt, muss das Feld Roboter\_id\_Roboter auf NULL gesetzt werden. Da im Funktionskopf nur ein Integer übergeben wird, wird über eine if-else-Abfrage die Variable @robo auf NULL gesetzt, wenn die, an die Funktion übergebene roboterid 0 ist. Nach dem Setzen der Variablen (Zeile 1-3) wird in Zeile 4 festgelegt welche Tabelle ein Update erfahren soll. Mit dem Set-Befehl aus Zeile 5 werden die beiden Felder aus Zeile 6 und 7 überschrieben. Dabei wird beim Überschreiben des Fremdschlüssels Roboter\_id\_Roboter die Beziehung eines Roboters zu einem Parkplatz geändert oder gekappt. In Zeile 8 und 9 wird über den Primärschlüssel ausgewählt, welche Zeile der Tabelle ein Update bekommt.

Listing 8.2: MySQL-Befehl: Update Parkplatz

```
1 SET @Parkplatz=ID;
2 SET @Status=status;
3 SET @robo=roboterid;
4 UPDATE vpj.parkplatz
5 SET
6     Status=@Status,
7     Roboter_id_Roboter=@robo
8 WHERE
9     id_Parkplatz=@Parkplatz;
```

Die Update Funktionen werden insbesondere für Tabellen genutzt in denen eine feste, durch die reale Welt vorgegebene, Anzahl an Zeilen vorhanden ist. Also zum Beispiel die Roboter. Die Roboter werden nicht mehr oder weniger, es ändern sich allerdings die einzelnen Werte der Eigenschaften.

### 8.7.3. Setter

Im Folgenden wird beispielhaft ein MySQL-Befehl aus einer Setter-Funktion in Qt beschrieben: Eine Setter-Funktion erzeugt einen neuen Eintrag in der Datenbank.

## 8. Datenbank

Das Listing 8.3 enthält die MySQL-Befehle zum Anlegen eines neuen Auftrags in der Datenbank und den dazugehörigen Werkstücken. Mit der Abfrage in den Zeilen 1-4 wird die höchste Auftrags ID ermittelt, damit der nächste Auftrag die nächst höhere ID bekommt und es keine Konflikte bei den Primärschlüsseln gibt. Mit dem INSERT INTO Befehl aus Zeile 5 wird eine neue Zeile in der Tabelle vpj.auftrag angelegt. Die Zeile erhält die Werte aus Zeile 8. Mit dem Befehl aus Zeile 9-12, wird nun eine neues Werkstück angelegt und über das Feld auftrag\_id\_auftrag mit dem Auftrag verknüpft. Da das Feld auftrag\_id\_auftrag nicht NULL sein darf, muss, bevor ein Werkstück in die Datenbank geschrieben wurde, immer erst der entsprechende Auftrag angelegt werden.

Listing 8.3: MySQL-Befehl: Update Parkplatz

```
1  SELECT
2    MAX(id_auftrag)
3  FROM
4    vpj.auftrag;
5  INSERT INTO
6    vpj.Auftrag
7  VALUES
8    (maxID, 0);
9  INSERT INTO
10   vpj.Werkstueck
11  VALUES
12    (werkstueck, 0, produktionsprozess, maxID);
```

Die Setter werden insbesondere für die Tabellen mit dynamischer Größe genutzt. Beispielhaft sei hier die Tabelle mit den Aufträgen genannt, die während der Programmlaufzeit die ganze Zeit über erweitert werden kann. Die Setter werden auch zu Programmstart genutzt, um die Datenbank initial zu befüllen.

# 9. Fertigungsüberwachung

Die Fertigungsüberwachung kontrolliert die Fertigung der Werkstücke und ermöglicht eine Verfolgung der Werkstücke während des Fertigungsprozesses. Dazu nutzt die sie RFID-Schreib-Lese-Köpfe, die über ein Interface durch Profibus-DP mit dem Fertigungsrechner verbunden sind und speichert die ermittelten Timestamps in der im Kapitel 8 beschriebenen Datenbank ab. Die Fertigungsüberwachung befindet sich ebenso wie die Datenbank auf dem Fertigungsrechner und ist als Soft-SPS ausgeführt. Für die Programmierung wird CODESYS genutzt. Für den Zugriff auf die Datenbank wird, wie in Abschnitt 8.5 ausgeführt, das Tool SQL4Automation genutzt.

## 9.1. Konzept

Es soll eine Werkstückverfolgung realisiert werden, so dass zu jedem Werkstück geprüft werden kann wann es an welcher Station eingeloggt oder ausgeloggt wurde. Dazu wird jedem Werkstück, wenn es bei der Auftragseingabe virtuell erzeugt und in der Datenbank abgelegt wird, eine eindeutige ID zugewiesen. Das Konzept zur Werkstückverfolgung sieht vor das Werkstück-Tag beim verlassen des Rohteillagers mit dieser ID zu beschreiben. Die Schreib-Lese-Köpfe der folgenden Stationen des Werkstücks lesen nun beim Ankommen und beim Verlassen der Station den Tag des Werkstücks aus und legen einen Timestamp an, der die Zeit, die Station und ob es sich um eine An- oder Abmeldung handelt, speichert. Dieser Timestamp wird mit dem dazugehörigen Werkstück verknüpft. Ist nun die ID des Werkstücks bekannt, können aus der Datenbank alle dazugehörigen Timestamps ausgelesen werden und es lässt sich so der Weg des Werkstücks nachverfolgen.

Die Schreib-Lese-Köpfe arbeiten dabei durchgehend und das Programm erkennt automatisch ob sich ein Werkstück unter den Schreib-Lese-Köpfen befindet. Eine direkte oder indirekte Kommunikation zu Gewerk 2 ist somit nicht nötig.

## 9.2. Programmstruktur

Das Programm ist so organisiert, das verschiedene Programmteile quasi parallel ablaufen. Dazu sind die einzelnen Programmteile in Funktionsbausteine ausgelagert. Die Instanzen dieser Funktionsbausteine werden in der Main oder in den jeweiligen Instanzen aufgerufen. So lässt sich das Programm strukturieren. In Abbildung 9.1 ist die Struktur der Funktionsbausteine und deren Instanzen dargestellt. Dabei ist immer der Instanzenname und in Klammern der Name des Funktionsbausteins angegeben. Türkis hinterlegt sind die Stellen, wo ein Funktionsbaustein

## 9. Fertigungsüberwachung

global instanziert wurde und innerhalb der lokalen Instanzen genutzt wird. Nähe-

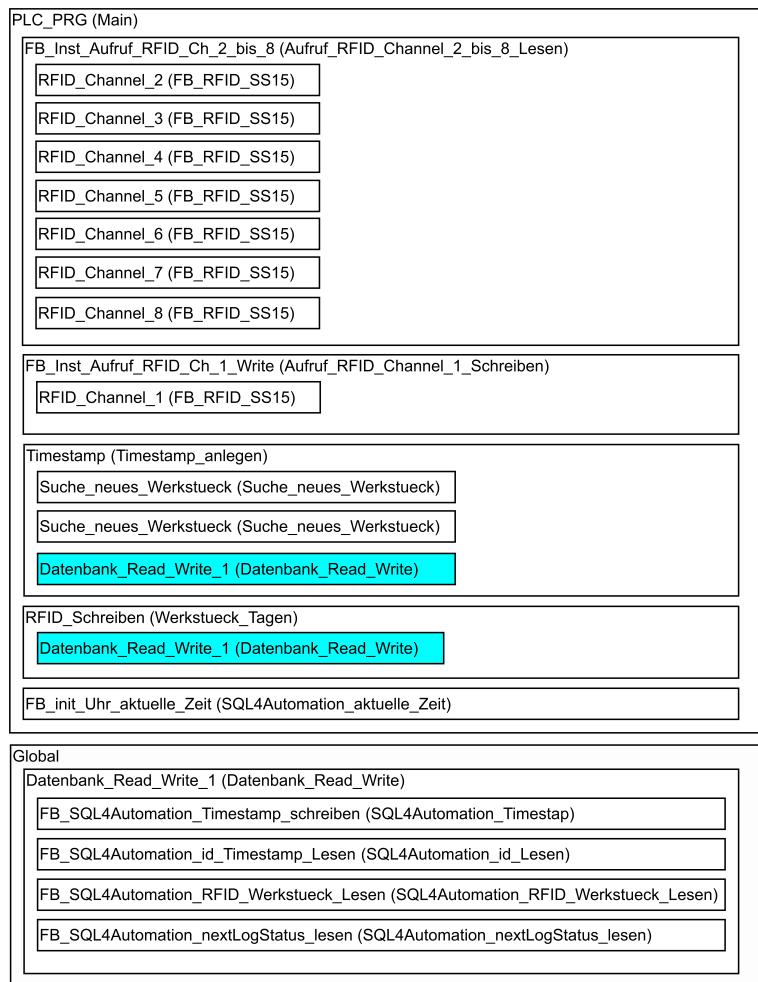


Abbildung 9.1.: Übersicht über die Funktionsbausteine und ihre Instanzen

re Informationen zu den einzelnen Funktionsbausteinen sind in Abschnitt 9.4 zu finden. Innerhalb der Funktionsbausteine ist das Programm überwiegend durch Zustandsautomaten organisiert. Dies ermöglicht ein strukturiertes Programm und den grafischen Entwurf mithilfe eines UML-Zustandsdiagramms.

Aufgeteilt ist das Programm in drei Funktionsteile. Der eine Teil sorgt für das Auslesen und Beschreiben der RFID-Tags. Dabei wird einerseits ein Abbild der aktuell eingelesenen Werte der RFID-Tags unter den Lesegeräten der Stationen 2 bis 8 erstellt und andererseits dem Schreib-Lese-Kopf an Station 1 der Wert zugewiesen, mit der der nächste RFID-Tag beschrieben werden soll. Dieser Teil befindet sich in den Funktionsbausteinen „Aufruf\_RFID\_Channel\_2\_bis\_8\_Lesen“ und „Aufruf\_RFID\_Channel\_1\_Schreiben“ und wird in den Abschnitten 9.4.2 und 9.4.3 ausführlich erläutert.

Der zweite Teil sorgt für die Kommunikation zwischen der Soft-SPS und der Datenbank und befindet sich im global instanzierten Funktionsbaustein „Datenbank\_Read\_Write“. Dieser Funktionsbaustein wird im Abschnitt 9.4.7 näher beschrieben.

## 9. Fertigungsüberwachung

Der letzte Teil sorgt für die Kommunikation zwischen den beiden anderen Teilen. Die eingelesenen Werte der RFID-Lesegeräte und die aus Datenbankabfragen gewonnen Werte werden ausgewertet und daraus Anweisungen an die anderen Programmteile generiert. Dieser Programmteil ist in den Funktionsbausteinen „werkstueck\_taggen“ und „Timestamp\_anlegen“ realisiert. Diese Funktionsbausteine werden in den Abschnitten 9.4.6 und 9.4.5 beschrieben.

### 9.3. Schnittstelle der RFID-Schreib-Lese-Köpfen

An den Stationen 1 bis 8 werden RFID-Schreib-Lese-Köpfe der Firma Turck eingesetzt. Diese RFID-Schreib-Lese-Köpfe sind an das Modulare Interface BL20 angeschlossen, welches ebenfalls von der Firma Turck ist. Das eingesetzte Interface BL20 hat 8 Kanäle für 8 Schreib-Lese-Köpfe. Über diese RFID-Schnittstelle sind die Schreib-Lese-Köpfe an das BL20 Interface angeschlossen. Zur Einbindung des Interface in die Feldbusebene können verschiedene Gateways modular ausgewählt werden. In diesem Fall wurde sich für die Einbindung durch Profibus-DP entschieden. Durch eine Profibus-DP-Masterkarte im Fertigungsrechner werden die Schreib-Lese-Köpfe an den Fertigungsrechner angeschlossen. Beim Bussystem Profibus-DP handelt es sich um ein Master-Slave-System. Die auf dem Fertigungsrechner als Soft-SPS realisierte Steuerung ist der Master und das RFID-Interface der Slave. In Abbildung 9.2 ist der Aufbau der Schnittstelle zwischen Fertigungsrechner und den Schreib-Lese-Köpfen dargestellt.[RFI19]

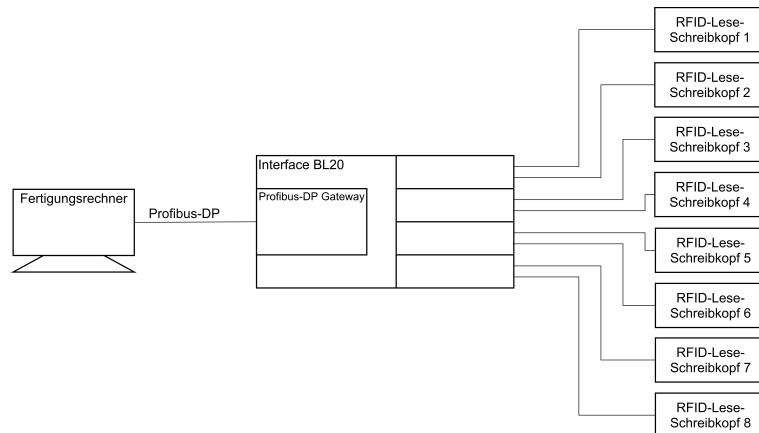


Abbildung 9.2.: Interface BL20 zur Einbindung der Schreib-Lese-Köpfe

### 9.4. Programmteile

Im folgenden Abschnitt wird die Funktion der einzelnen Funktionsbausteine und der Main, die im Projekt verwendet wurden, beschrieben. Eine Übersicht über die verwendeten Funktionsbausteine liefert die Abbildung 9.1. In Abschnitt 9.2 ist beschrieben, wie die einzelnen Funktionsbausteine ins Gesamtkonzept integriert sind.

### **9.4.1. Main PLC – PRG**

Die Main teilt sich in zwei wesentliche Teile auf, die durch eine if-else-Abfrage getrennt werden. Zu Beginn der Main befindet sich ein kurzer Programmteil, in dem die aktuelle Uhrzeit ermittelt wird. Der Programmteil ist in Abschnitt 9.4.9 näher beschrieben. Ist dieser Programmteil abgeschlossen und die aktuelle Uhrzeit ermittelt, wechselt die Main in den else-Zweig. In diesem Teil der Main werden die Funktionsbausteine „Aufruf\_RFID\_Channel\_2\_bis\_8\_Lesen“, „Aufruf\_RFID\_Channel\_1\_Schreiben“, „Werkstueck\_Tagen“ und "Timestamp\_anlegen“ aufgerufen. Im unteren Teil der Main befindet sich noch ein kurzer Programmabschnitt zur Messung der Zykluszeit.

### **9.4.2. FB Aufruf\_RFID\_Channel\_2\_bis\_8\_Lesen**

Dieser Funktionsbaustein liest die RFID-Leseköpfe an den Stationen 2 bis 8 aus. Es handelt sich hierbei um eine Erweiterung des im Staterkit zur Verfügung gestellten Funktionsbausteins „Aufruf\_RFID\_Channel\_1\_bis\_8“. Die Leseköpfe 2 bis 8 werden nur lesend betrieben. Der eigentliche Zugriff auf die RFID-Leseköpfe findet in den sieben Instanzen des auch mit dem Staterkit zur Verfügung gestellten Funktionsbausteins „FB\_RFID\_SS15“ statt. Auch die Initialisierung findet in diesen Instanzen statt und muss über die entsprechende Eingangsvariable des Funktionsbausteins gestartet werden .

Die Instanzen des Funktionsbausteins „FB\_RFID\_SS15“ werden jedes Mal wenn der Funktionsbaustein „Aufruf\_RFID\_Channel\_2\_bis\_8\_Lesen“ aufgerufen wird, auch aufgerufen. Zu Beginn des Funktionsbausteins befindet sich ein Zustandsautomat, der die Initialisierung der Instanzen des „FB\_RFID\_SS15“ durchführt und den Lesebetrieb festlegt. Um den Code kürzer und übersichtlicher zu gestalten, befindet sich der Zustandsautomat in einer for-Schleife, wobei jeder Schleifendurchlauf einem Lesekopf entspricht. Die entsprechenden Variablen für die Initialisierung befinden sich dazu in einem Array. Der Zustandsautomat für die Initialisierung ist in Abbildung 9.3 dargestellt.

Ist die Initialisierung abgeschlossen, werden die RFID-Schreib-Lese-Köpfe kontinuierlich ausgelesen. Ist ein RFID-Tag unter dem Lesegerät, wird der Wert des Tags in einem Array gespeichert. Dabei entspricht die Stelle im Array der Station, an der der Tag eingelesen wurde. Ist kein Werkstück unter dem Schreib-Lese-Kopf wird in das Array eine 0 geschrieben. Da der Wert, den die Schreib-Lese-Kopf zurück liefern, immer dem des zuletzt eingelesenen RFID-Tags entspricht, wird mithilfe des Parameters TFR überprüft, ob sich aktuell ein RFID-Tag unter dem Schreib-Lese-Kopf befindet. Der TFR-Parameter gibt an, ob sich ein RFID-Tag innerhalb des Lesebereichs befindet. Befindet sich kein RFID-Tag unter dem Schreib-Lese-Kopf wird der Wert auf 0 gesetzt. Befindet sich ein RFID-Tag darunter wird gewartet bis der RFID-Tag ausgelesen und der Wert ins Array geschrieben wurde.

Die Informationen werden auf den RFID-Tags als Bytes gespeichert. Zur weiteren Verarbeitung werden die Bytes in einem Integer zusammengefasst. Dies über-

## 9. Fertigungsüberwachung

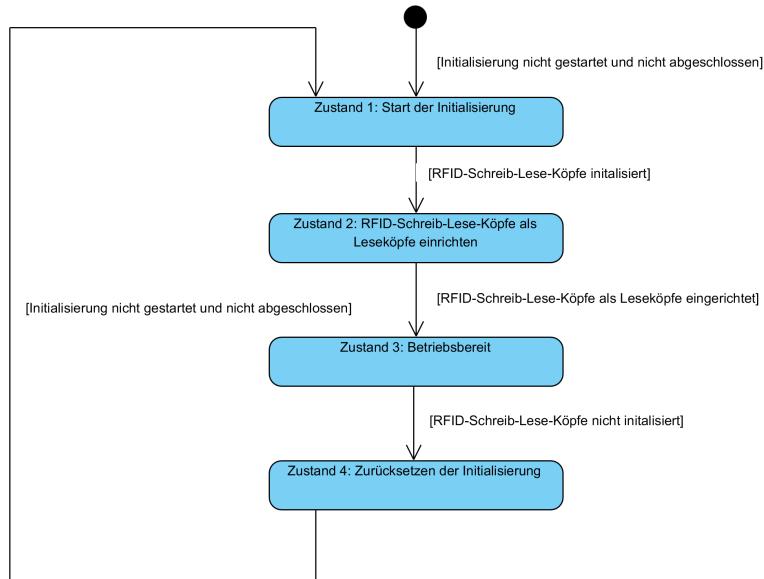


Abbildung 9.3.: UML-Zustandsdiagramm der Initialisierung der RFID-Leseköpfe 2 bis 8

nimmt die selbst geschriebene Funktion ByteToInt(), die als Eingangsvariable ein Byte-Array und als Ausgangsvariable eine Integervariable hat.

### 9.4.3. FB Aufruf \_ RFID \_ Channel \_ 1 \_ Schreiben

Dieser Funktionsbaustein organisiert das Beschreiben der RFID-Tags an Station 1. Im Aufbau ist er dem in Abschnitt 9.4.2 beschriebenen Funktionsbaustein ähnlich. Allerdings befindet sich hier nur eine Instanz des Funktionsbausteins „FB\_RID\_SS15“ für den Schreib-Lese-Kopf an Station 1. Der Zustandsautomat entspricht dem in Abbildung 9.3. Lediglich der Zustand 2 unterscheidet sich, da dort der Schreib-Lese-Kopf nicht als Lesekopf sondern als Schreibkopf eingerichtet wird.

Über die Rückgabewerte Busy und Done wird detektiert, ob gerade ein RFID-Tag neu beschrieben wurde. Wird ein Schreibvorgang erkannt, wird die Globale-Variable „WRITE\_RIDI\_DONE“ gesetzt. Diese Variable wird im Funktionsbaustein „Timestamp\_anlegen“ aus Abschnitt 9.4.6 beim Suchen nach einem neuen Werkstück verwendet. Dort wird sie auch zurückgesetzt. Damit sie nach dem Zurücksetzen nicht sofort wieder gesetzt wird, sollte sich das Werkstück noch unter dem Schreibkopf befinden, wird mittels eines Timers 6s gewartet. Das bedeutet, dass auch nur alle 6s ein Werkstück aus dem Lager geholt werden kann. Die Roboter sind allerdings nicht in der Lage innerhalb von 6s zwei Werkstücke aus dem Lager zu holen, so dass es zu keinen Problemen führt.

### 9.4.4. FB FB \_ RFID \_ SS15

Der Funktionsbaustein „FB\_RID\_SS15“ wurde mit dem Starterkit zur Verfügung gestellt. Der im Programm verwendete Funktionsbaustein entspricht fast

## 9. Fertigungsüberwachung

komplett dem Funktionsbaustein aus dem Starterkit, weshalb er hier nicht weiter erläutert wird. Nur für das Errorhandling sind wenige Zeilen Code hinzugekommen.

Ist der Rückgabewert „Channel\_X\_ERROR“ True wird ein Reset durchgeführt bis der Fehler behoben wurde. Zum Testen der Fehleranfälligkeit befindet sich ein Zähler im Errorhandling.

### 9.4.5. FB Werkstueck\_Tagen

Immer wenn ein Roboter ein Werkstück aus dem Lager holen soll, wird von der Fertigungsplanung in die Datenbank geschrieben um welches Werkstück es sich handelt. Dieser Funktionsbaustein fragt durchgehend die Datenbank ab und ermittelt so welche die nächste ID ist die an Station 1 auf den RFID-Tag des Werkstücks geschrieben werden muss. In Abbildung 9.4 ist das dazugehörige Zustandsdiagramm dargestellt.

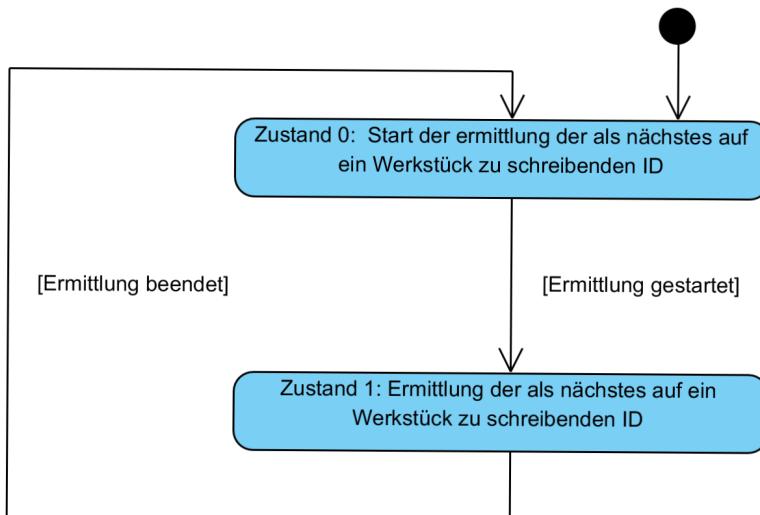


Abbildung 9.4.: UML-Zustandsdiagramm des FB Werkstueck\_taggen

### 9.4.6. FB Timestamp\_anlegen

Der Funktionsbaustein „Timestamp\_anlegen“ ist zuständig für die Auswertung des RFID-Arrays, in dem die aktuell eingelesenen Werkstück ID's gespeichert sind. Des Weiteren ermittelt der Funktionsbaustein alle für das Anlegen des Timestamps erforderlichen Werte und schreibt mit Hilfe des Funktionsbausteins „Datenbank\_Read\_Write“ den Timestamp in die Datenbank. In der Abbildung 9.5 ist das Zustandsdiagramm des Funktionsbaustein dargestellt.

Zunächst wird im Zustand 0 mit Hilfe der Funktion „Suche\_neues\_Werkstueck“ nach einem neu unter den RFID-Schreib-Lesekopf geschobenes Werkstück gesucht. Zu Beginn wird allerdings zunächst geprüft, ob an Station 1 ein Werkstück neue beschrieben wurde. Dazu wird die globale Variable „WRITE\_RFID\_DONE“

## 9. Fertigungsüberwachung

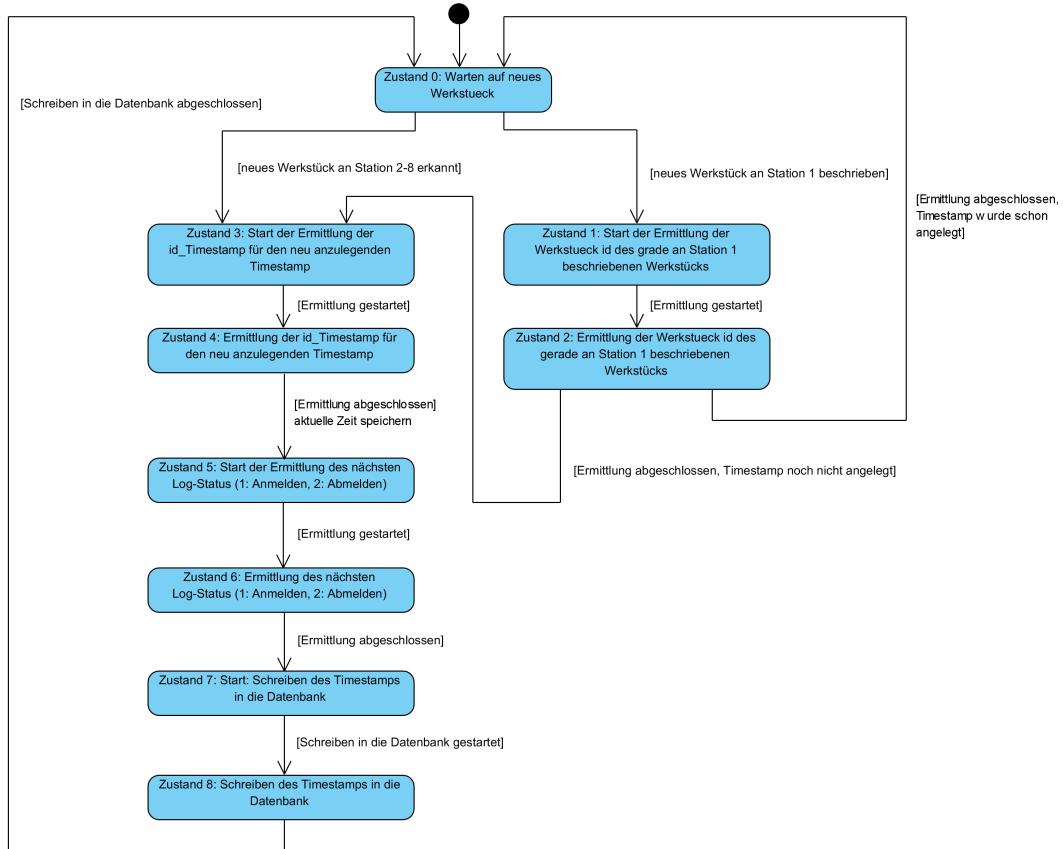


Abbildung 9.5.: UML-Zustandsdiagramm des FB Timestamp\_anlegen

ausgewertet und ggf. zurückgesetzt. Wurde kein Werkstück neu beschrieben, wird überprüft, ob sich seit dem letzten Aufruf der Funktion ein Wert im RFID-Array geändert hat. Dazu wird ein Vergleich mit einer Kopie des Arrays aus dem letzten Aufruf durchgeführt. Wird eine Änderung erkannt und es handelt sich nicht um eine Änderung auf den Wert 0, wird die Station an der die Änderung aufgetreten ist und die ID des erkannten Werkstücks von der Funktion zurückgegeben. Ebenso wird die geänderte Kopie des RFID-Arrays zurückgegeben.

Wurde ein Werkstück an Station 1 neu beschrieben, so muss zunächst mit einer Datenbankabfrage aus der Tabelle „taggen“ die ID des beschriebenen Werkstücks ermittelt werden. Dies geschieht im Zustand 2. Der Zustand 1 startet die Ermittlung. Für einen Zugriff auf die Datenbank werden immer zwei Zustände benötigt, einer in dem signalisiert wird das eine Abfrage stattfinden soll. Beginnt die Abfrage, wird dieser Zustand verlassen und im darauf folgenden Zustand wird gewartet bis die Abfrage abgeschlossen ist. Diese zwei Zustände ergeben sich aus dem Zusammenspiel mit dem global instanzierten Funktionsbaustein "Datenbank\_Read\_Write", der in Abschnitt 9.4.7 näher beschrieben ist.

Wurde die ID des neu beschriebenen Werkstücks ermittelt, liegen genau die gleichen Informationen, nämlich die ID des Werkstücks und die Station an der es erkannt wurde, wie bei einem Werkstück, dass neu unter einen Lesekopf der Stationen 2-8 geschoben wurde, vor. Für beide möglichen Fälle geht es weiter mit Zustand 3 und 4, in denen die nächste ID für den Timestamp aus der Datenbank

## 9. Fertigungsüberwachung

ermittelt wird. Beim Übergang von Zustand 4 in Zustand 5 wird die aktuelle Zeit als Unix-Timestamp gespeichert. Anschließend wird im Zustand 5 und 6 der nächste Log-Status ermittelt. Der Log-Status beschreibt, ob es sich um eine Anmeldung oder eine Abmeldung handelt. Zum Ermitteln des Log-Status wird der letzte Eintrag für das jeweilige Werkstück in der Tabelle „Timestamp“ in der Datenbank ermittelt und der Status dementsprechend gesetzt. Besonders werden die Fälle an Station 1 und an Station 8 behandelt, da dort nur An- beziehungsweise Abmeldungen möglich sind.

Nachdem nun die ID des Werkstücks, die Station, die ID des nächsten Timestamps, der nächste Log-Status und die aktuelle Zeit bekannt sind, werden die Daten in den Zuständen 7 und 8 als Timestamp in der Datenbank abgelegt und zu Zustand 0 zurückgekehrt.

### 9.4.7. FB Datenbank\_Read\_Write

Der Funktionsbaustein „Datenbank\_Read\_Write“ sorgt für einen geordneten Zugriff auf die Datenbank. Der eigentliche Zugriff auf die Datenbank findet in den vier Funktionsbausteinen „SQL4Automation\_id\_Lesen“, „SQL4Automation\_nextLogStatus\_Lesen“, „SQL4Automation\_Timestap“ und „SQL4Automation\_RFID\_Werkstueck\_Lesen“ statt. Da die Funktionsbausteine die selben globalen Variablen nutzen und mehrere Zyklen für einen Zugriff benötigen, dürfen nicht mehrere der Funktionsbausteine gleichzeitig versuchen auf die Datenbank zuzugreifen. Weil aber Anfragen an die Datenbank von zwei verschiedenen Funktionsbausteinen aus erfolgen, muss sichergestellt sein, dass diese Anfragen nicht gleichzeitig erfolgen. Dafür wird dieser global instanzierte Funktionsbaustein genutzt. Wenn aus einem der lokal instanziierten Funktionsbausteine heraus eine Anfrage gestellt wird, prüft der Funktionsbaustein „Datenbank\_Read\_Write“, ob diese Anfrage gerade möglich ist. Sollte die Anfrage nicht an der Reihe sein, wartet der Funktionsbaustein der die Anfrage gestellt hat, bis der Funktionsbaustein „Datenbank\_Read\_Write“ die Anfrage zulässt und die Abfrage der Datenbank startet.

Der Funktionsbaustein „Datenbank\_Read\_Write“ prüft in dem Zustandsautomaten aus Abbildung 9.6, ob eine Anfrage für den jeweiligen Datenbankzugriff vorliegt. Liegt keine Anfrage vor, geht er zum nächsten Zustand, in dem auf eine Anfrage geprüft wird. Liegt eine Anfrage vor startet er einen Datenbankzugriff und geht für die Dauer des Zugriffs in einen Zustand, der die Anfrage ausführt. An die anfragenden Funktionsbausteine meldet er zurück, ob ein Datenbankzugriff gestartet wurde und ob der gestartete Zugriff abgeschlossen wurde. Während sich der Funktionsbaustein in einem ausführenden Zustand befindet, werden weiter Anfrage der anderen Funktionsbausteine abgelehnt.

### 9.4.8. FB SQL4Automation

Für jeden Zugriff auf die Datenbank ist ein separater Funktionsbaustein zuständig. Aufgerufen werden die Funktionsbausteine durch den global instanziierten

## 9. Fertigungsüberwachung

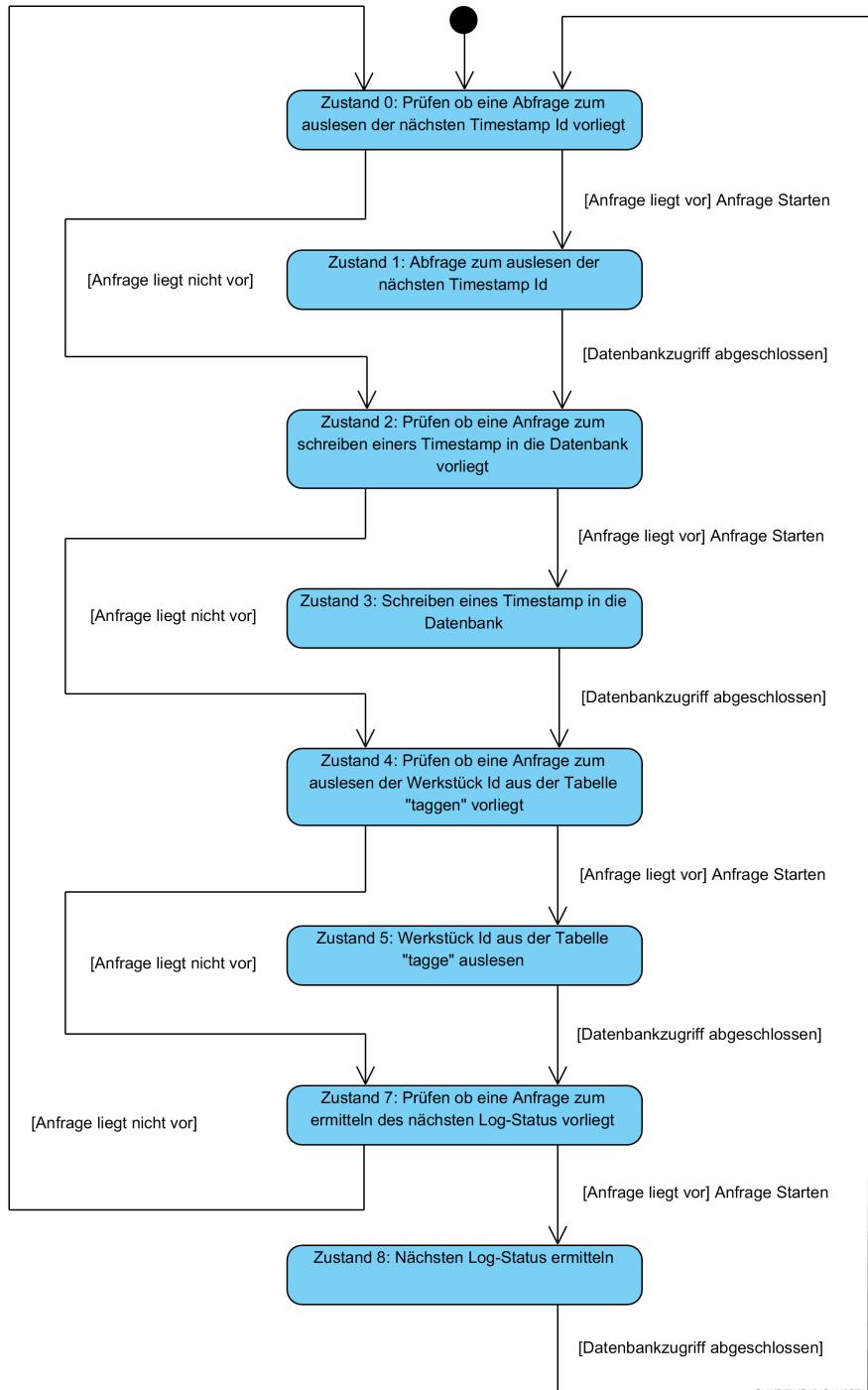


Abbildung 9.6.: UML-Zustandsdiagramm des FB Datenbank\_Read\_Write

Funktionsbaustein „Datenbank\_Read\_Write“, der in Abschnitt 9.4.7 beschrieben ist. Die Funktionsbausteine entsprechen in großen Teilen dem mit dem Staterkit zur Verfügung gestellten Funktionsbaustein „SQL4Automation\_2015“. Angepasst wurde für die jeweiligen Funktionsbausteine der MySQL-Befehl zum Lesen oder Schreiben in der Datenbank. Des Weiteren wurde bei Lesebefehlen ein zusätzlicher Rückgabewert für das Ergebnis der Abfrage und die entsprechende Konvertierung des Rückgabe-Strings der Datenbank in den richtigen Datentyp hinzugefügt. Ebenso wurde eine RückgabevARIABLE hinzugefügt, die zurückmeldet, ob die Anfrage an die Datenbank abgeschlossen ist. Wie schon in Kapitel 9.4.7

## 9. Fertigungsüberwachung

beschrieben, nutzen alle Funktionsbausteine, die den Zugriff auf die Datenbank durchführen, die selben globalen Variablen. Da ein Zugriff mehrere Zyklen dauert, ist sicherzustellen, dass nicht zwei Funktionsbausteine zur selben Zeit eine Anfrage an die Datenbank stellen.

### 9.4.9. Einstellen der Systemzeit

Da die Systemzeit nach jedem Start der Soft-SPS auf den default-Wert 1. Januar 1970 0:00:00 Uhr gesetzt wird und das manuelle Einstellen der Systemzeit bei jedem Neustart der Soft-SPS, beziehungsweise des aufgespielten CODESYS-Programms, ungenau und aufwendig ist, wird die aktuelle mitteleuropäische Zeit zu Beginn des Programms automatisch ermittelt. Zur Ermittlung der Zeit für die Timestamps wird der Funktionsbaustein „RTC“ genutzt. Dieser wird zu Programmbegin im Funktionsbaustein „Timestamp\_anlegen“ einmal initialisiert. Für die Initialisierung wird die aktuelle Zeit im Datentyp DT benötigt. Die aktuelle Zeit wird zu Programmbegin in der Main über die Datenbank ermittelt.

Wie in Abbildung 8.3 zu sehen, gibt es in der Datenbank eine Tabelle mit dem Namen Zeit, diese Tabelle hat nur eine Zeile in der es einmal den Primärschlüssel „id\_Zeit“ und das Feld „aktuelle\_Zeit“ gibt. Bei Start des Programms wird in der Main zunächst der Funktionsbaustein „SQL4Automation\_aktuelle\_Zeit“ aufgerufen. Dabei handelt es sich um den einzigen Funktionsbaustein der auf die Datenbank zugreift und nicht über den Funktionsbaustein „Datenbank\_Read\_Write“ aufgerufen wird. Da das Ermitteln der aktuellen Zeit erst abgeschlossen sein muss, bevor der restliche Programmteil startet, besteht keine Gefahr eines gleichzeitigen Zugriffs. Der Funktionsbaustein „SQL4Automation\_aktuelle\_Zeit“ führt die zwei in den Listings 9.1 und 9.2 dokumentierten SQL-Befehle aus. Der Befehl aus Listing 9.1 aktualisiert die Zeit im Feld „aktuelle\_Zeit“ in der Datenbank. Dazu wird ein Update der einzigen Zeile in der Tabelle ausgeführt. Besonders ist hierbei der SET-Befehl, der dem Feld „aktuelle\_Zeit“ den Rückgabewert der MySQL-Funktion NOW() zuweist (Zeile 3). Die Funktion NOW() liefert die aktuelle Zeit, abhängig von der konfigurierten Zeitzone. Mit dem Befehl in Listing 9.2 wird diese abgefragt.

Listing 9.1: MySQL-Befehl: Zeit aktualisieren

```
1 UPDATE vpj.Zeit
2 SET
3     aktuelle_Zeit = NOW()
4 WHERE
5     id_ZEIT = 1;
```

Listing 9.2: MySQL-Befehl: Zeit abfragen

```
1 SELECT
2     aktuelle_Zeit
3 FROM
4     vpj.Zeit
5 WHERE
6     id_ZEIT = 1;
```

Mit Hilfe von String-Operationen wird der zurückgelieferte String so umgebaut, dass er mit Hilfe eines Cast vom Datentyp String in den Datentyp DT (Date

## *9. Fertigungsüberwachung*

and Time) umgewandelt werden kann. Ist die Ermittlung der aktuellen Zeit abgeschlossen, startet das eigentliche Programm.

# 10. Zusammenfassung

Qt als neues System ist sehr gut geeignet.

# Abbildungsverzeichnis

1.1. Gesamtkonzept von Gewerk 1 . . . . .	2
2.1. Signale und Slots in Qt Quelle: <a href="https://doc.qt.io/qt-5/signalsandslots.html">https://doc.qt.io/qt-5/signalsandslots.html</a> . . . . .	4
2.2. State Machine Beispiel Quelle: <a href="https://doc.qt.io/qt-5/statemachine-api.html">https://doc.qt.io/qt-5/statemachine-api.html</a> . . . . .	6
3.1. Sequenzdiagramm . . . . .	9
3.2. Positionskodierung . . . . .	10
4.1. Klassendiagramm . . . . .	14
4.2. Visualisierung des Rückgabewertes der Funktionen GetNextUnfinishedProzessstep() und GetActualProzessstep() . . . . .	25
4.3. Diagramm Fahrweganalyse . . . . .	28
4.4. Diagramm Stationsanalyse . . . . .	28
4.5. Zustandsdiagramm der Auftragsplanung . . . . .	32
5.1. VPJ-Logo . . . . .	41
5.2. Übersicht über die Bereiche innerhalb der Visualisierung . . . . .	42
5.3. Übersicht Visualisierung . . . . .	43
5.4. Visualisierung Live-View . . . . .	43
5.5. Robotino Darstellung bei verschiedenen Hinderniserkennungen . . . . .	44
5.6. Roboter in verschiedenen Stati (vlnr: Greifer offen, Greifer geschlossen, Defekt, Einhorn) . . . . .	44
5.7. Visualisierung Parkplatz . . . . .	45
5.8. Visualisierung Ladestation . . . . .	45
5.9. Stationen mit verschiedenen Arbeitsplatzstati . . . . .	46
5.10. Auftragsfortschritt . . . . .	47
5.11. Auftrag ein- und ausgeblendet . . . . .	47
5.12. Auftragsitem (links) und Prozessitem (rechts) . . . . .	48
5.13. Pause Button gedrückt . . . . .	50
5.14. Tab: Log View . . . . .	51
5.15. Tab: Manual Control . . . . .	53
5.16. Tab: Info Area . . . . .	54
5.17. Tab: Hard-Code . . . . .	54
5.18. Vergleich Visualisierung - Normal VS Hard-Code Modus . . . . .	55
5.19. Batterie und Statusanzeige . . . . .	57
5.20. Roboterstatus-LED blinkend . . . . .	58
5.21. Visualisierung - Prozesseingabe . . . . .	59
5.22. Visualisierung - Auftragsvergabe . . . . .	59
5.23. Tooltip eines Werkstücks . . . . .	61

## *Abbildungsverzeichnis*

5.24. Tooltip eines Prozessitem . . . . .	62
6.1. Ablaufdiagramm des simulierten Roboters . . . . .	63
7.1. Fertigungsrechner und Schnittstellen . . . . .	66
8.1. Entität-Relationship-Diagramm . . . . .	68
8.2. Schematische Darstellung des Datenbankzugriffs . . . . .	72
8.3. Grafisches Modell der Datenbank . . . . .	73
9.1. Übersicht über die Funktionsbausteine und ihre Instanzen . . . . .	79
9.2. Interface BL20 zur Einbindung der Schreib-Lese-Köpfe . . . . .	80
9.3. UML-Zustandsdiagramm der Initialisierung der RFID-Leseköpfe 2 bis 8 . . . . .	82
9.4. UML-Zustandsdiagramm des FB Werkstueck_taggen . . . . .	83
9.5. UML-Zustandsdiagramm des FB Timestamp_anlegen . . . . .	84
9.6. UML-Zustandsdiagramm des FB Datenbank_Read_Write . . . . .	86

# Literaturverzeichnis

- [Cas17] Stack Exchange Inc. *Regular cast vs. static\_cast vs. dynamic\_cast*, 2017. Stand am 09.05.2019:  
<https://stackoverflow.com/questions/28002/regular-cast-vs-static-cast-vs-dynamic-cast.html>.
- [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).
- [HSS01] Andreas Heuer, Gunter Saake, and Kai-Uwe Sattler. *Datenbanken kompakt*. mitp-Verlag, 2001.
- [Jar16] Helmut Jarosch. *Grundkurs Datenbankentwurf Eine beispielorientierte Einführung für Studierende und Praktiker*. Springer Vieweg, 2016.
- [Qt 19a] The Qt Company Ltd. *About Qt*, 2019. Stand am 09.05.2019:  
[https://wiki.qt.io/About\\_Qt](https://wiki.qt.io/About_Qt).
- [Qt 19b] The Qt Company Ltd. *QListWidget Class*, 2019. Stand am 09.05.2019: <https://doc.qt.io/qt-5/qlistwidget.html>.
- [Qt 19c] The Qt Company Ltd. *QSqlDatabase Class*, 2019. Stand am 09.05.2019: <https://doc.qt.io/qt-5/qsqldatabase.html>.
- [Qt 19d] The Qt Company Ltd. *QUdpSocket Class*, 2019. Stand am 09.05.2019: <https://doc.qt.io/qt-5/qudpsocket.html>.
- [Qt 19e] The Qt Company Ltd. *Signals & Slots*, 2019. Stand am 09.05.2019: <https://doc.qt.io/qt-5/signalsandslots.html>.
- [Qt 19f] The Qt Company Ltd. *The State Machine Framework*, 2019. Stand am 09.05.2019:  
<https://doc.qt.io/qt-5/statemachine-api.html>.
- [RFI19] Hans Turck GmbH & Co. KG. *Handbuch RFID-System BL ident Planen und Projektieren*, 2019.

## A. Inhalt der CD

- Dieses Dokument als PDF „VPJ.pdf“
- Die Qt Projektdateien des Hauptprogramms, inklusive aller zum Start benötigten Dateien als Verbundprojekt.zip
- Die Qt Projektdateien des Starterkit, inklusive aller zum Start benötigten Dateien als Verbundprojekt\_Starterkit.zip
- Alle Abbildungen der Arbeit im Ordner „Abbildungen“
- MySQL Modell der MySQL Workbench „Verbundprojekt\_model.mwb“
- CODESYS Projektdatei „Fertigungsrechner.project“
- Demovideo des Gesamtprojekts „VPJ\_Demovideo.mp4“

## **Erklärung zur selbstständigen Bearbeitung einer Arbeit**

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(\\$ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

*Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI*

## **Erklärung zur selbstständigen Bearbeitung der Arbeit**

Hiermit versichern wir,

Name: \_\_\_\_\_

Vorname: \_\_\_\_\_

Name: \_\_\_\_\_

Vorname: \_\_\_\_\_

dass wir die vorliegende Arbeit mit dem Thema:

### **Verbundprojekt 2018/2019**

ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt haben. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Die Bearbeitung und Dokumentation der Themen aus Kapitel 2 bis 6, sowie alle Tätigkeiten auf dem Fertigungsplanungsrechner erfolgten dabei durch Daniel Friedrich. Die Bearbeitung und Dokumentation der Themen aus Kapitel 7 bis 9, sowie alle Tätigkeiten auf dem Fertigungsrechner erfolgten durch Jan-Henrik Meyer.

---

Ort

Datum

---

Unterschrift im Original

---

Unterschrift im Original