

# Konzeption und Implementierung eines Schwarmverhaltens von mobilen Kleinrobotern anhand eines Verfolgungsszenarios

## STUDIENARBEIT

für die Prüfung zum  
Bachelor of Science  
des Studiengangs Informatik  
Studienrichtung Angewandte Informatik  
an der

Dualen Hochschule Baden-Württemberg Karlsruhe

22. Mai 2017

Name	Manuel Bothner	Simon Lang
Matrikelnummer	8359139	6794837
Kurs	TINF14B2	TINF14B2
Ausbildungsfirma	1&1 Internet SE Brauerstr. 48 76135 Karlsruhe	ifm ecomatic GmbH Im Heidach 18 88079 Kressbronn am Bodensee
Betreuer	Prof. Hans-Jörg Haubner	

## Erklärung

(gemäß §5(3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. 9. 2015)

Ich versichere hiermit, dass ich die Studienarbeit meiner Studienarbeit mit dem Thema: „Konzeption und Implementierung eines Sachwurmverhaltens von mobilen Kleinrobotern anhand eines Verfolgungsszenarios“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

---

Ort, Datum

Unterschrift

---

Ort, Datum

Unterschrift

## Abstract

## Zusammenfassung

---

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Ausgangslage . . . . .	2
1.2 Zielsetzung . . . . .	2
<b>2 Theoretische und Technische Grundlagen</b>	<b>3</b>
2.1 Robotik . . . . .	3
2.1.1 Roboter . . . . .	3
2.1.2 Mobile Roboter . . . . .	4
2.1.3 Sensorik . . . . .	5
2.1.4 Sensordatenverarbeitung & Steuerung . . . . .	8
2.1.5 Antriebsarten . . . . .	10
2.2 LEGO MINDSTORMS . . . . .	13
2.2.1 Das EV3-System . . . . .	13
2.2.2 Der EV3-Stein (Steuereinheit) . . . . .	14
2.2.3 Motoren . . . . .	15
2.2.4 Sensoren . . . . .	15
2.2.5 Programmierung . . . . .	17
2.3 Application (App) Entwicklung . . . . .	19
2.3.1 Native Apps . . . . .	19
2.3.2 Web Apps . . . . .	19
2.3.3 Hybride Apps . . . . .	20
2.3.4 Plattformübergreifende Apps . . . . .	20
2.3.5 Xamarin . . . . .	21
2.3.6 Mono . . . . .	25
2.3.7 .NET Framework . . . . .	25
2.4 TCP-Kommunikation . . . . .	27
2.4.1 Gundlegendes . . . . .	27
2.4.2 Nagle-Algorithmus . . . . .	28
2.4.3 Kommunikationsablauf . . . . .	28
2.4.4 Socket-Programmierung . . . . .	30
2.5 Schwarmverhalten . . . . .	31
2.5.1 Allgemein . . . . .	31
2.5.2 Vorbilder aus dem Tierreich . . . . .	31
2.5.3 Szenarien . . . . .	32
<b>3 Konzeption</b>	<b>35</b>
3.1 Anforderungsdefinitionen . . . . .	35
3.2 Softwarearchitektur . . . . .	36

3.3	Steuerung	38
3.4	Szenarien	38
3.5	Use Cases	40
3.5.1	Connection	40
3.5.2	Synchronization	42
3.5.3	Szenario	44
3.5.4	Exception	46
3.6	Kommunikation	47
<b>4</b>	<b>Implementierung</b>	<b>50</b>
4.1	Kommunikation	50
4.2	App	55
4.2.1	Workflow	56
4.2.2	Graphical User Interface (GUI)	57
4.2.3	Buisnesslogic	62
4.3	Backend	67
4.4	Aufbau	67
4.4.1	Server	67
4.4.2	Client	69
4.4.3	Interpreter	69
4.4.4	Device & Szenario Controller	70
4.4.5	Die GUI	71
4.5	Roboter	72
4.5.1	Threads	72
4.5.2	Klassen	73
4.5.3	Steuerung	76
4.5.4	Datenerfassung	76
<b>5</b>	<b>Evaluation</b>	<b>77</b>
<b>6</b>	<b>Ausblick</b>	<b>79</b>

## Abkürzungsverzeichnis

**AOT** Ahead of Time.

**API** Application Programming Interface.

**App** Application.

**ARM** Acorn RISC Machines.

**CLI** Common Language Infrastructure.

**CLR** Common Language Runtime.

**CPU** Central Processing Unit.

**CSS** Cascading Style Sheets.

**DLL** Dynamic Linked Library.

**GPS** Global Positioning System.

**GUI** Graphical User Interface.

**HTML** Hypertext Markup Language.

**IL** Intermediate Language.

**IP** Internet Protocol.

**JIT** Just-in-Time.

**JSON** JavaScript Object Notation.

**JVM** Java Virtual Machine.

**KI** Künstliche Intelligenz.

**MVVM** Model-View-ViewModel.

**OSI** Open Systems Interconnection.

**PCL** Portable Class Library.

**SSH** Secure Shell.

**TCP** Transmission Control Protocol.

**UDP** Unified Datagram Protocol.

**UI** User Interface.

**UML** Unified Modeling Language.

**VDI** Verein Deutscher Ingenieure.

**VM** Virtual Machine.

**XAML** Extensible Application Markup Language.

**XML** Extensible Markup Language.

## Glossar

**Application Programming Interface** Programmierschnittstelle, die die Anbindung von Software ermöglicht.

**C#** Objektorientierte Programmiersprache mit dem Schwerpunkt auf Typsicherheit.

**Eclipse** Programmierumgebung für diverse Programmiersprachen und Frameworks.

**EV3** Kleinroboter der LEGO Mindstorm Serie.

**Framework** Rahmen zur Programmierung mit verschiedenen Softwarekomponenten.

**Java** Objektorientierte Programmiersprache für plattformübergreifende Anwendungen.

**JavaFX** Framework zur Erstellung von Java Anwendungen.

**JavaScript** Skriptsprache zur Entwicklung dynamischer Internetseiten.

**leJOS** Java System zur Programmierung von LEGO Mindstorm.

**Objective-C** Auf C basierte objektorientierte Programmiersprache.

**SQLite** Bibliothek zur Erstellung einer lokalen relationalen Datenbank.

**Template** Vorlage zur Implementierung.

**TypeScript** Objektorientierte Programmiersprache für Webentwicklung.

---

## Abbildungsverzeichnis

1	Vergleich Offene und Geschlossene Steuerung . . . . .	8
2	Einachsige Lenkung . . . . .	10
3	Einachsige Lenkung . . . . .	11
4	Dreirad-Antrieb . . . . .	11
5	Einachsige Lenkung . . . . .	11
6	Einachsige Lenkung . . . . .	12
7	Zentrale Komponenten des EV3-Systems . . . . .	13
8	App-Entwicklung [? ] . . . . .	19
9	Xamarin [? ] . . . . .	21
10	Shared Project [? ] . . . . .	22
11	Portable Class Library [? ] . . . . .	22
12	Unterstützung Portable Class Library [? ] . . . . .	22
13	Design Xamarin.Forms [? ] . . . . .	23
14	SignInPage in XAML . . . . .	23
15	Xamarin.Forms Layouts [? ] . . . . .	24
16	Xamarin.Forms Pages [? ] . . . . .	24
17	Mono [? ] . . . . .	25
18	.NET Framework Ausführung [? ] . . . . .	26
19	Common Language Runtime [? ] . . . . .	26
20	TCP Verbindungsaufbau . . . . .	29
21	TCP Datenaustausch . . . . .	29
22	TCP Verbindungsabbau . . . . .	30
23	Fischschwarm . . . . .	31
24	Termitenschwarm . . . . .	32
25	Roboter bilden Formen [? ] . . . . .	33
26	Roboter bilden Formen [? ] . . . . .	34
27	Industrieroboter [? ] . . . . .	34
28	Softwarearchitektur . . . . .	37
29	Steuerung . . . . .	38
30	Logo . . . . .	39
31	Mockup Connection . . . . .	40
32	Use Case Connection . . . . .	41
33	Mockup Synchronization . . . . .	42
34	Use Case Synchronization . . . . .	43
35	Use Case Update . . . . .	43
36	Mockup Szenario . . . . .	44
37	Use Case Szenario . . . . .	45

38	Use Case Spectator . . . . .	45
39	Use Case Exception . . . . .	46
40	Aufbau Commands . . . . .	50
41	Kommunikation . . . . .	51
42	Kommandos . . . . .	52
43	Identifikation . . . . .	52
44	Devices . . . . .	53
45	Scenarios . . . . .	53
46	JsonConverter . . . . .	54
47	Device JsonConverter . . . . .	54
48	Projektstruktur . . . . .	55
49	Model-View-ViewModel [? ] . . . . .	56
50	Wechsel auf neue Seite [? ] . . . . .	57
51	Wechsel auf alte Seite [? ] . . . . .	57
52	StackLayout [? ] . . . . .	57
53	SignIn Page . . . . .	58
54	Hauptseiten . . . . .	59
55	List Page . . . . .	60
56	Szenario Page . . . . .	61
57	Timeout der Verbindung . . . . .	62
58	Verbindungsauflauf . . . . .	62
59	Speicherung der Internet Protocol (IP)-Adresse . . . . .	63
60	Definition von SQLite . . . . .	63
61	Hinzufügen der Roboter . . . . .	64
62	Erstellung des Szenario . . . . .	64
63	Abbruch eines Szenario . . . . .	65
64	Neigungssensor . . . . .	65
65	Neues Kommando . . . . .	66
66	<code>open()</code> -Methode der Sever-Klasse im Backend . . . . .	67
67	<code>listen()</code> -Methode der Sever-Klasse im Backend . . . . .	68
68	Ausschnitt der <code>parse()</code> -Methode der Interpreter-Klasse im Backend . . . . .	69

## Tabellenverzeichnis

1	Einordnung von Sensoren . . . . .	6
2	Eigenschaften der EV3-Motortypen . . . . .	15
3	Eigenschaften der EV3-Motortypen . . . . .	18
4	JavaScript Object Notation (JSON) Kommando Connection . . . . .	47
5	JSON Kommando Synchronization . . . . .	47
6	JSON Kommando Scenario . . . . .	48
7	JSON Kommando Exception . . . . .	48
8	JSON Kommando Control . . . . .	49
9	JSON Kommando Position . . . . .	49

# 1 Einleitung

Heutzutage werden viele Arbeitsschritte im Agrar-, Industrie- und Dienstleistungssektor von Computern und Roboter verrichtet, da diese eine effizientere Arbeit leisten und weniger Kosten als Menschen verursachen. Diese Technologien erfuhrten in den letzten Jahren einen immer stärkeren Wandel durch laufende technische Innovationen, die deren Arbeitsablauf verbessern und somit produktiver gestalten. Eine dieser Technologien stellt das theoretische Konzept eines Schwarmverhaltens dar, welches Unternehmen zur Kooperation verschiedener Computer und Roboter einsetzen. Dies ermöglicht die gegenseitige Unterstützung der Komponenten und somit einen geteilten Arbeitsablauf, um die jeweiligen Stärken zu nutzen. Diese Verhaltensstrukturen stammen meist aus der Tierwelt, wie am Beispiel von Fischschwärmern, Ameisen oder Bienen, wobei jedes Individuum des Schwarms seine Aufgaben für das Überleben des Schwarms erfüllt.

In diesem Projekt werden die grundlegenden Verhaltensstrukturen von in Schwärmern lebenden Tieren zur Umsetzung eines Verfolgungsszenarios, wie am Beispiel eines autonom fahrenden Autos, aufgegriffen. Dabei werden feste Regeln anhand von Benutzerszenarios definiert, auf welche der Roboterschwarm entsprechend der Nutzerhandlungen reagiert. Dies wird durch ein zentrales Kommunikationssystem umgesetzt, an welches alle teilnehmenden Komponenten angeschlossen sind.

Die Einsatzgebiete dieses Projektes sind dabei entsprechend groß und kann überall eingesetzt werden, wo technische Komponenten für ein gemeinsames Produkt zusammenarbeiten müssen. Beispiele hierfür ist die Optimierung von Produktionsanlagen, oder Verkehrsführungen in Form von autonom fahrenden Autos.

## 1.1 Ausgangslage

Die Ausgangslage des Projektes stellen verschiedene Computer und Roboter ohne intelligentes Kommunikationssystem dar, welche einen Mehrwert durch eine Dienstleistung oder die Bearbeitung eines Produktes erwirtschaften. In diesem Projekt wird dies durch den Einsatz von LEGO Mindstorm EV3 Roboter dargestellt, welche mittels Struktur orientierter Programmierung verwendet werden können. Diese verfügen über verschiedene Schnittstellen sowie Sensorik, um mit ihrer Umwelt interagieren zu können und stellen damit die Basis eines Schwarmes dar.

- LEGO Mindstorm EV3
- LEGO Mindstorm Sensorik
- Kabellose Netzwerkschnittstelle

## 1.2 Zielsetzung

Das Ziel dieser Studienarbeit stellt die Implementierung eines Schwarmverhaltens mit dem Fokus auf ein Verfolgungsszenario zwischen Kleinrobotern dar. Dabei soll ein zentrales Kommunikationssystem aufgebaut werden, auf dessen Grundlage die Komponenten miteinander interagieren. Die Roboter enthalten hierbei Grundfunktionen, die über Kommandos des Kommunikationssystems angesprochen werden, um die definierten Benutzerszenarios auszuführen und den Roboter zu bewegen.

- Zentrales Kommunikationssystem
- Grundlegende Steuerungsfunktionen
- Abbildung von Schwarmverhalten
- Darstellung der aktuellen Daten

---

## 2 Theoretische und Technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen, die zur Umsetzung des Projektes nötig sind beschrieben. Dazu gehören Grundlagen der Robotik, Kommunikation sowie Platzhalter kommt noch Text

Platzhalter kommt noch Text Darüber hinaus werden theoretische Grundlagen der Kommunikation und des Verhaltens von Schwärmen gegeben.

### 2.1 Robotik

Die Robotik beschäftigt sich mit dem Entwurf, der Konstruktion sowie der Programmierung, Steuerung und dem Betrieb von Robotern.<sup>1</sup> Dabei umfasst die Robotik eine Vielzahl von Fachgebieten wie der Elektrotechnik, dem Maschinenbau, der Informatik sowie der Biologie und der Medizin.

#### 2.1.1 Roboter

Was im Kontext der Robotik unter einem Roboter zu verstehen ist, ist gar nicht so einfach darzulegen, da es in Tat keine allgemein anerkannte Definition dieses Begriffs gibt, die seiner üblichen Verwendung entspricht.<sup>2</sup>

Auch ohne eine vollkommen allgemeingültige und präzise Beschreibung eines Roboters zu sein, soll hier die **Verein Deutscher Ingenieure (VDI)**-Richtline 2860 einen Eindruck darüber vermitteln, was in der Robotik mit Roboter gemeint ist.

Die **VDI**-Richtline 2860 von 1990 definiert einen Roboter wie folgt:<sup>2</sup>

*„Ein Roboter ist ein frei und wieder programmierbarer, multifunktionaler Manipulator mit mindestens drei unabhängigen Achsen, um Materialien, Teile, Werkzeuge oder spezielle Geräte auf programmierten, variablen Bahnen zu bewegen zur Erfüllung der verschiedensten Aufgaben.“*

Auch wenn diese Definition einige grundlegenden Eigenschaften eines Roboters darlegt, beschreibt diese den Begriff des Roboters im industriellen Kontext und trifft hauptsächlich auf stationäre Industrieroboter zu, wie sie in der Automobilfertigung beispielsweise als Schweiß- oder Lackierroboter verwendet werden. Die genannten programmierten Bahnen sind dort möglich, da die Arbeitsprozess und Umgebung auf den Roboter zugeschnitten und vollständig bekannt sind.<sup>2</sup>

Im Gegensatz dazu, trifft dieser Aspekt auf mobile Roboter nicht zu, da sich diese meist in einer unbekannten, unstrukturierten und dynamischen Umgebung bewegen.

---

<sup>1</sup>[vgl. ?, Definition Robotik]

<sup>2</sup>[vgl. ?, Mobile Roboter, Seite 2]

## 2.1.2 Mobile Roboter

Mobile Roboter bewegen sich selbstständig durch eine sich meist ständig ändernde Umwelt. All ihre Aktionen sind somit von ihrer aktuellen Umgebung abhängen, die in ihrer konkrete Ausprägung erst zum Zeitpunkt der Aktionsausführung im Detail bekannt ist. Dieser grundsätzliche Unterschied zu stationären Robotern macht es unerlässlich das mobile Roboter ihre Umgebung ständig mittels Sensoren selbstständig erfassen, die Sensordaten auswerten und auf Grundlage dessen ihre nächsten Aktionen planen.<sup>3</sup>

Hinter dem Begriff des mobilen Roboters verbergen sich eine Vielzahl unterschiedlicher mobiler Systeme, welche sich hinsichtlich ihrer Gestaltung, Arbeitsweise und ihrer Einsatzszenarien unterscheiden. Im Folgenden werden einige Beispiele für mobile Roboter dargelegt:

- **Shakey:** Shakey ist ein mobiler Roboter der von 1966 bis 1972 an der am Stanford Research Institute entwickelt wurde. Seine Entwicklung leistete wichtige Beiträge für die Robotik sowie in der **Künstliche Intelligenz (KI)**-Forschung im Bereich der Handlungsplanung und dem selbständigen Lernen.<sup>4</sup>
- **Spirit & Opportunity:** Spirit & Opportunity sind zwei baugleiche Roboter die im Jahr 2003 von der NASA zum Mars geschickt wurden um den Himmelskörper zu erkunden. Die beiden Erkundungsroboter sind Radfahrzeuge mit flexiblem Fahrgestell, verfügen über eine Panoramakamera sowie Sensoren zur Untersuchung des Erdbodens und Gesteins. Obwohl die Roboter in Bezug auf ihrer grundsätzlichen Aktionen von der Erde aus ferngesteuert werden, ist eine autonome Steuerung, welche auf kurzfristige unerwartet Ereignisse, wie das Wegrutschen von Rädern reagiert aufgrund der langen Signallaufzeiten unverzichtbar. Die Roboter waren für eine Lebensdauer von 90 Marstagen ausgelegt, übertrafen diese aber bei weitem mit mehr als dem dreißig fachen.<sup>5</sup>
- **Stanley:** Stanley ist ein vollständig autonomer Roboter der 2005 am Grand Challenge Wettkampf teilnahm und diesen gewann. Bei diesem Wettbewerb mussten Fahrzeuge ohne Eingriff von Menschen eine festgelegte, jedoch nicht markierte Strecke von rund 213 km von einem definierten Start- zu einem definierten Zielpunkt zurücklegen. Die Strecke führte durch die Mojave-Wüste in den USA. Bei Stanley handelt es sich um einen modifizierter VW Touareg, dem Sensoren zur Umgebungs-wahrnehmung und Bordrechner zur Bearbeitung des Kontrollprogramms eingebaut wurden. Stanleys wichtigste Umgebungssensoren waren mehrere Laserscanner und eine Kamera. Stanley meisterte die 213 km lange Strecke welche unter anderem

<sup>3</sup>[vgl. ? , Mobile Roboter, Seite 2]

<sup>4</sup>[vgl. ? , Mobile Roboter, Seite 5 f.]

<sup>5</sup>[vgl. ? , Mobile Roboter, Seite 8 f.]

---

durch felsige oder sandige Bereiche sowie durch Wasserläufe führte in knapp unter 7 Stunden.<sup>6</sup>

Allein diese drei Beispiele zeigen wie sehr sich mobile Roboter im Aufbau, Einsatzort und Aufgabe unterscheiden. Neben reinen Forschungs- sowie wissenschaftlich-technischen Gründen Mobile Roboter zu bauen, sind diese auch aus wirtschaftlicher Perspektive interessant und haben gerade in den letzten Jahren an Marktpotenzial gewonnen.

Zu den Vertretern mobiler Roboter im kommerziellen Bereich zählen zum Beispiel Service-Roboter, Erkundungsroboter und Bergungsroboter, humanoide Roboter sowie Haushaltsroboter.

### 2.1.3 Sensorik

Um mit der Umgebung interagieren zu können, müssen mobile Roboter diese wahrnehmen, dazu dienen Sensoren. Diese ermöglichen es dem Roboter Informationen über seine Umwelt und über seinen Zustand zu sammeln um darauf Aufbauen seine nächsten Interaktionsschritte zu planen.

**Klassifizierung** Sensoren lassen sich anhand zweier Aspekten klassifizieren, zum einen hinsichtlich des Objektes über welches sie Informationen liefern (die Umwelt oder den Roboter selbst) und anderseits hinsichtlich ihrer Arbeitsweise.<sup>7</sup>

- **Propriozeptive Sensoren** – Diese Art der Sensoren bestimmen eine Messgröße des Roboters selbst und haben keine „Kontakt“ zur Umwelt z.B. Bestimmung der Lage Aufgrund eines Neigungssensors.
- **Exterozeptive Sensoren** – Im Gegensatz zu den propriozeptive Sensoren gewinnen diese Sensoren Informationen aus Messgrößen der Umwelt beispielsweise die Bestimmung der Orientierung in Bezug auf die Umwelt.
- **Aktive Sensoren** – Aktive Sensoren senden aktive Energie in ihre Umwelt aus und Erfassen anschließend die zurückkehrenden Signale wie dies beispielsweise ein Ultraschallsensor tut.
- **Passive Sensoren** – Diese Sensoren senden nicht aktiv aus sondern erfassen ausschließlich die von der Natur aus vorhandenen Signale wie z.B. das einfallende Licht durch eine Kamera.

Die folgenden Tabelle zeigt beispielhaft die Einordnung einiger Sensoren:

---

<sup>6</sup>[vgl. ? , Mobile Roboter, Seite 9 f.]

<sup>7</sup>[vgl. ? , Mobile Roboter, Seite 24]

	Aktive Sensoren	Passive Sensoren
Propriozeptive Sensoren	–	Inkrementalgeber, Neigungssensor, Gyroskop
Exterozeptive Sensoren	Ultraschallsensor, Laserscanner, Infrarotsensor, Radar	Kontaktsensor, Kompass, Kamera, GPS

Tabelle 1: Einordnung von Sensoren

**Eigenschaften** Neben der Arbeitsweise gibt es noch weitere Eigenschaften die maßgeblich beeinflussen wann, wie und für welchen Zweck ein jeweiliger Sensor eingesetzt wird. Zu den wichtigsten Eigenschaften zählen:<sup>8</sup>

- **Messbereich** – Jeder Sensor hat einen bestimmten Bereich, in dem die gemessenen Daten valide sind d.h. der Bereich in dem die Messabweichungen innerhalb der festgelegten Grenzen bleibt.
- **Dynamik** – Die Dynamik beschreibt das Verhältnis von Ober- zu Untergrenze des Messwerts.
- **Auflösung** – Die Auflösung beschreibt wie granular eine physikalische Größe ermittelt werden kann d.h sie gibt den kleinsten messbaren Unterschied zweier Messwerte an.
- **Linearität** – Unter der Linearität eines Sensors versteht man die Abhängigkeit des Messwerts von der tatsächlichen Größe.
- **Messfrequenz** – Die Messfrequenz gibt an wie viele einzelne Messungen innerhalb eines bestimmten Zeitraums der Sensor liefert.

Da jegliche Messung fehlerbehaftet ist, weisen auch die von Sensoren ermittelte Messwerte gewisse Fehler auf.<sup>9</sup>

Damit die Robotersteuerung darauf Rücksicht nehmen kann müssen die entsprechenden Kenngrößen der Sensoren bekannt sein. Folgende Größen sind mit Bezug auf Sensorfehler von Bedeutung:<sup>10</sup>

- **Empfindlichkeit** – Wie groß eine Wertänderung der Ausgangsgröße sein muss damit dies der Sensor registriert, wird als Empfindlichkeit bezeichnet. Eine hohe Empfindlichkeit hat oft eine hohe Störanfälligkeit zur Folge.

<sup>8</sup>[vgl. ? , Mobile Roboter, Seite 26 f.]

<sup>9</sup>[vgl. ? , Mobile Roboter, Seite 27]

<sup>10</sup>[vgl. ? , Mobile Roboter, Seite 27 f.]

- **Messfehler** – Als Messfehler oder absoluten Fehler bezeichnet man die Differenz des gemessenen Wertes  $m$  und des tatsächlichen Wertes.
- **Genauigkeit** – Die Genauigkeit oder der relative Fehler ist der prozentuale Wert einer Abweichung in Bezug zum tatsächlichen Wert.

**Kenngrößen** In der Robotik finden zahlreiche Sensoren Anwendung, um es einem mobilen Systems zu ermöglichen verschiedene Kenngrößen zu ermitteln. Im Folgenden werden einiger dieser und die dazu geeigneten Sensoren vorgestellt.

- **Bewegungsmessung** – Eine Möglichkeit der Bewegungsmessung ist die Drehwinkelmessung. Diese ermöglicht die Erfassung der Drehungen von Rädern oder anderen rotierenden Elementen eines mobilen Roboters. Dies ermöglicht es mittels Odometrie die Geschwindigkeit und Orientierung eines mobilen Roboters zu bestimmen, siehe Abschnitt (2.1.4). Zur Drehwinkelmessung dienen Impuls- sowie Inkrementalgeber welche mechanisch, photo-elektrisch und elektro-magnetisch arbeiten.<sup>11</sup> Eine weitere Möglichkeit der Bewegungsmessung ist die Beschleunigungsmessung. Dabei werden Bewegungen über die Massenträgheit bei Beschleunigungen gemessen und über die Zeit integriert, weshalb man auch von Trägheitsmessung spricht.<sup>12</sup> Dazu werden heute meist piezo-elektronische Gyroskope verwendet welche Dreh- und Linearbeschleunigungen bezüglich aller drei Raumachsen erfassen.<sup>13</sup>
- **Ausrichtungsmessung** – Neben der Bestimmung der geografische Position eines Roboters mittels Beschleunigung- und Drehratensensoren ist auch dessen Ausrichtung bzw. Lage im Raum eine wichtige Größe. Zur Bestimmung der Ausrichtung werden Kompassse sowie Inklinometer verwendet, welche den Neigungswinkel zur Erdanziehungsrichtung bestimmen.<sup>14</sup>
- **Entfernungsmessung** – Die Entfernungsmessung dient der Bestimmung des Abstands zu Objekten z.B. zur Kollisionsvermeidung.  
Es gibt verschiedene Arten von Sensoren zur Entfernungsmessung viele basieren auf dem Prinzip der Laufzeitmessung, wie Ultraschallsensoren oder Laserentfernungs-messer. Darüber hinaus gibt es auch Sensoren, die mittels Triangulation (Inrarot-sensor) oder dem Verhältnis zwischen ausgesandter und zurückkehrender Energie (Radar) die Entfernung zu Objekten bestimmen.<sup>15</sup>

Neben diesen elementaren Kenngrößen gibt es zahlreiche weitere die für ein entsprechendes mobile System von Bedeutung sind. Dies ist beispielsweise eine globale Positionsbestim-

<sup>11</sup>[vgl. ? , Mobile Roboter, Seite 28 ff.]

<sup>12</sup>[vgl. ? , Mobile Roboter, Seite 31]

<sup>13</sup>[vgl. ? , Mobile Roboter, Seite 32 ff.]

<sup>14</sup>[vgl. ? , Mobile Roboter, Seite 32 f.]

<sup>15</sup>[vgl. ? , Mobile Roboter, Seite 36 f.]

mung mittels **Global Positioning System (GPS)** oder die Dedektion von Objekten und geografischen Strukturen durch Kameras oder 2D-/3D-Laserscanner.

#### 2.1.4 Sensordatenverarbeitung & Steuerung

Generell unterscheidet man zwischen zwei grundlegenden Arten der Steuerung, einer offenen und geschlossenen. Das typische Kontrollprogramm eines Roboters in der Automatisierung läuft in einer offenen Steuerung d.h. es werden keine Sensordaten aus der Umgebung erfasst bzw. berücksichtigt. Dies ist dort möglich, da sich dort Roboter auf vordefinierten Bahnen und in eine auf sie zugeschnittenen Umgebung bewegen.

Prozesssteuerungen dieser Art haben ihre Grenzen da, wo sich ein Roboter in einer unstrukturierten Umgebung bewegt und von Ereignissen oder Parametern abhängt, die nicht kontrollierbar bzw. vorab nicht bekannt sind.<sup>16</sup> Mobile Roboter verfügen über eine geschlossene Steuerung bei der eine Rückkopplung der Umgebung durch Sensordaten erfolgt. In einem iterativen Verfahren werden die von den Sensoren ermittelten Umgebungsdaten bei der Planung von Steuerbefehle einbezogen, die resultierenden Aktionen ausgeführt was ggf. die Umgebung verändert, woraufhin der Regelkreis erneut beginnt.

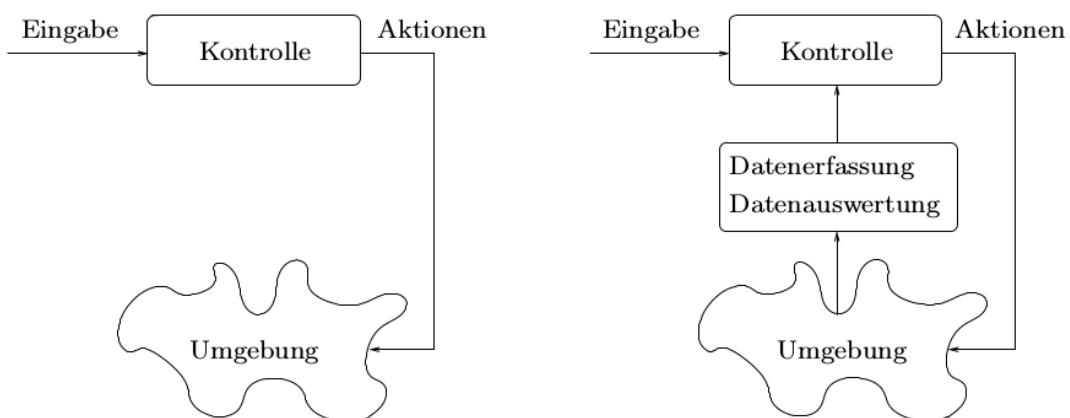


Abbildung 1: Links: Offene Steuerung. Rechts: Geschlossene Regelung mit Rückkopplung [?]

Die Sensordatenverarbeitung hat die Aufgabe, die durch die Sensoren erfassten Daten zu verarbeiten, aufzubereiten und sie für komplexere Anwendungen nutzbar zu machen. Dazu gehören die Filterung der Daten, Extraktion bestimmter Merkmale sowie weiterführende Berechnung von Kenngrößen und Informationen.<sup>17</sup>

**Odometrie & Koppelnavigation** Die Odometrie bezeichnet eine Methode zur Berechnung von Position und Orientierung eines mobilen Systems anhand von Bewegungsdaten. Räder basierte Systeme benutzen dafür die Anzahl der Radumdrehungen, während laufende Systeme die Anzahl ihrer Schritte verwenden. Ist beispielsweise der Raddurchmesser

<sup>16</sup>[vgl. ?, Mobile Roboter, Seite 3 f.]

<sup>17</sup>[vgl. ?, Mobile Roboter, Seite 67]

bekannt, so kann über die Anzahl  $n$  der Radumdrehungen (welche z.B. mittels Inkrementalgebern erfasst wird) zwischen zwei Messzeitpunkten und dem Radumfang  $\pi \cdot d$ , wobei  $d$  den Raddurchmesser darstellt, die zurückgelegte Wegstrecke  $\Delta s$  berechnet werden:

$$\Delta s = \pi \cdot d \cdot n \quad (1)$$

Unter Koppelnavigation versteht man die Bestimmung der Position eines mobilen Systems relativ zu einem Referenzpunkt (Startposition) entweder durch die inkrementelle Integration der zurückgelegten Wegstrecken unter Berücksichtigung der Orientierung (Winkeländerung) oder der Geschwindigkeit des Systems unter Berücksichtigung der Winkelgeschwindigkeit.<sup>18</sup>

Ist der Ausgangspunkt eines Roboters, die zurückgelegte Strecke bzw. Geschwindigkeit sowie die Fahrtrichtung genau bekannt, kann die Navigationskomponente die Bewegung des Robotersystems über die Zeit integrieren und so die aktuelle Position und Orientierung des Roboters bestimmen.<sup>19</sup>

Im eindimensionalen Fall (keine Richtungsänderungen) ergibt sich aus dem Startpunkt  $s_{alt}$  und der zurückgelegten Wegstrecke  $\Delta s$  aus Gleichung (1), für die aktuelle Position  $s_{neu}$  die Gleichung:

$$s_{neu} = s_{alt} + \Delta s \quad (2)$$

Für eine zweidimensionale Bewegung in einem 2-dimensionalen kartesischen Bezugssystem in dem sich die Position eines Roboters als Koordinate durch x- und y-Punkt darstellen lässt ergibt sich die Position  $(x_n, y_n)$  und die Orientierung  $\theta_n$  zum Iterationsschritt (Messpunkt)  $n$  durch die Formeln:

$$x_n = x_0 + \sum_{i=1}^n \Delta s_i \cdot \cos(\theta_i) \quad (3)$$

$$y_n = y_0 + \sum_{i=1}^n \Delta s_i \cdot \sin(\theta_i) \quad (4)$$

$$\theta_n = \theta_0 + \sum_{i=1}^n \Delta \theta_i \quad (5)$$

Wobei  $(x_0, y_0)$  der Startpunkt und  $\theta_0$  die Startorientierung sowie  $\Delta s_i$  die zurückgelegte Wegstrecke und  $\Delta \theta_i$  die Richtungsänderung innerhalb eines Iterationsschritt darstellt.

Die Odometrie ist im Zusammenspiel mit der Koppelnavigation ein grundlegendes Navigationsverfahren für bodengebundene Fahrzeuge aller Art. Allerdings besteht ein wesentliches Problem darin, dass ein Robotersystem seine Bewegungen absolut präzise messen

<sup>18</sup>[vgl. ? , Mobile Robotik, Seite 98]

<sup>19</sup>[vgl. ? , Handbuch Robotik, Seite 107 f.]

können muss, damit die Odometrie bzw. Koppelnavigation richtige Werte liefert. Dies ist unter Umständen aufgrund von Problemen wie beispielsweise variierender Raddurchmesser dem Durchdrehen oder Gleiten der Räder erschwert wird.<sup>20</sup>

Wegen diesen Problemen finden sich kaum Roboternavigationssysteme, welche ausschließlich eine Koppelnavigation zur Bestimmung von Position und Orientierung verwenden. Es gibt zwar Navigationssysteme, welche die Koppelnavigation verwenden, jedoch beziehen diese noch weitere Sensorinformationen, wie beispielsweise Beschleunigungswerte zur Bestimmung der Position und Ausrichtung mit ein.<sup>21</sup>

### 2.1.5 Antriebsarten

Die entscheidende Eigenschaft des mobilen Roboters ist die Fähigkeit sich selbstständig durch seine Umwelt zu bewegen. Um dies zu realisieren gibt es eine Vielzahl unterschiedlichster Fortbewegungsarten, beispielsweise in Form von schreitenden, kriechenden oder krabbelnden, sowie fliegenden, schwimmenden oder tauchenden Robotern.<sup>22</sup>

Um den Umfang dieser Ausarbeitung überschaubar zu halten, beschränken wir uns auf die Darstellung von radgetriebenen Robotern.

Hinsichtlich Rad- und Achsenanzahl, Anordnung der Räder sowie dem Verhältnis zwischen angetrieben und freilaufenden, als auch lenkbaren zu unlenkbaren Rädern gibt es eine Vielzahl von Möglichkeiten den Antrieb von mobilen Robotern zu konzipieren. Die folgende Abschnitte geben einen Überblick über die gängigsten radbasierten Antriebskonzepte.<sup>23</sup>

**Differentialantrieb** Dieser Antrieb besteht aus zwei von einander unabhängig angetriebene, starren Rädern die sich auf einer Achse befinden. Zusätzlich gibt es meist ein oder auch mehrere passiv mitlaufenden Stützräder, die sich frei drehen können. Je nachdem, wie und in welchem Verhältnis zueinander die beiden Antriebsräder sich drehen, fährt der Roboter gerade aus, eine mehr oder weniger weite Rechts- oder Linkskurve oder dreht sich auf der Stelle.

Vorteile dieses Antriebs sind seine einfache Mechanik und gute Manövierbarkeit, welcher jedoch eine Radregelung des Antriebs in Echtzeit erfordert.

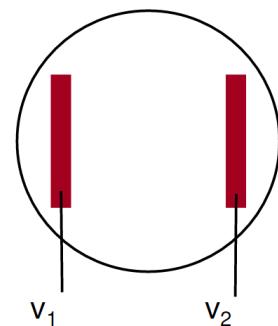


Abbildung 2: Einachsige Lenkung

<sup>20</sup>[vgl. ? , Handbuch Robotik, Seite 108]

<sup>21</sup>[vgl. ? , Mobile Robotik, Seite 115]

<sup>22</sup>[vgl. ? , Mobile Roboter, Seite 103]

<sup>23</sup>[vgl. ? , Mobile Roboter, Seite 107 f.]

**Einachsige Lenkung** Dieses Antriebsprinzip entspricht dem aus gängigen PKWs bekannten Konzept. Von den vier Rädern, von denen jeweils zwei an einer Achse angebracht sind, sind beide Räder einer Achse lenkbar. Für den Antrieb ist es möglich eine der beiden Achsen, als auch beide anzutreiben.

Der Vorteil dieses Antriebskonzeptes ist die gute Stabilität und die Möglichkeit der Trennung von Antrieb und Lenkung jedoch ist die Manövierbarkeit eingeschränkter als bei den anderen Antriebsarten.

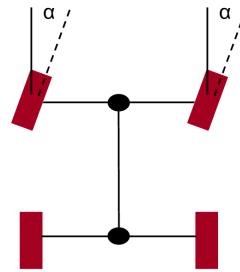


Abbildung 3: Einachsige Lenkung

**Dreirad-Antrieb** Bei diesem Antriebskonzept verfügt der mobile Roboter über zwei freilaufenden Räder auf einer Achse sowie einem einzelnen angetriebenen und gelenkten Rad, welches zentral vor den beiden anderen Rädern angebracht ist. Es ist auch möglich, dass ein Motor beide Hinterräder antreibt, und das Vorderrad allein der Lenkung dient.

Die einfache Mechanik dieser Antriebsform ist ein Vorteil, jedoch ist die Manövierbarkeit eingeschränkter als beim Synchro- oder Differentialantrieb und die Stabilität ist geringer, als bei der Einachsen Lenkung, die über ein zusätzliches Rad an der Vorderachse verfügt.

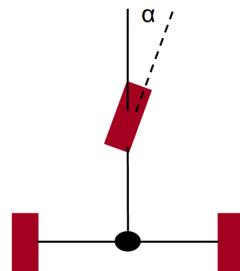


Abbildung 4: Dreirad-Antrieb

**Synchro-Antrieb** Der Synchro-Antrieb hat die selbe Radanordnung, wie der Dreirad-Antrieb, jedoch werden die drei Räder synchron angetrieben und sind auch nur synchron drehbar. Die Räder haben somit immer die selbe Ausrichtung und drehen sich auch mit der selben Geschwindigkeit. Eine Drehung des Roboters bzw. der auf dem Antrieb angebrachte Plattform ist nicht direkt möglich.

Durch die synchrone Lenkung und den synchronen Antrieb der Räder erfordert diese Antriebsform eine weitaus komplexere Mechanik, als die anderen Antriebsarten, garantiert jedoch einen stetigen Geradeauslauf und erfordert keine komplexe Regelung.<sup>24</sup>

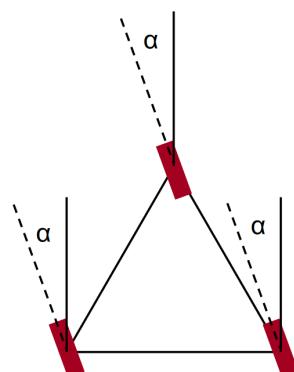


Abbildung 5: Einachsige Lenkung

<sup>24</sup>[vgl. ?, Mobile Roboter, Seite 107 f.]

**Omni-Antrieb** Beim Omni-Antrieb kommen omnidirektionale Räder wie Allseitenräder oder Mecanum-Räder zum Einsatz. Durch ihren speziellen Aufbau ist der Roboter in der Lage, sich sowohl auf der Stelle zu Drehen, als auch in alle Richtungen zu bewegen, ohne sich vorher drehen zu müssen. Dies verleiht dem Antrieb eine uneingeschränkte Beweglichkeit in jede Richtungen (x, y und  $\omega$ ).

Der Große Vorteil dieser Antriebsform ist die uneingeschränkte Beweglichkeit. Jedoch birgt diese Art des Antriebs eine gewisse mechanische Komplexität und erfordert eine aufwändige Steuerung.

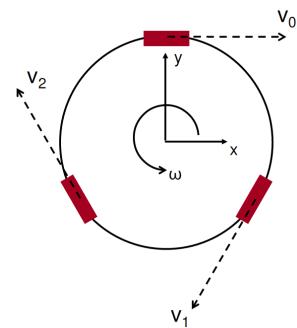


Abbildung 6: Einachsige Lenkung

## 2.2 LEGO MINDSTORMS

LEGO MINDSTORMS ist eine seit 1988 existierende Produktserie des Spielwarenherstellers LEGO.<sup>25</sup> LEGO MINDSTORMS ermöglicht das Bauen, Programmieren und Steuern verschiedener LEGO Roboter. Diese Roboter bestehen dabei aus gängigen LEGO Teilen die auch in anderen LEGO-Produkten Verwendung finden, sowie speziellen LEGO-Komponenten wie einer zentralen Steuereinheit, Motoren und Sensoren.

### 2.2.1 Das EV3-System

Der 2013 erschienene EV3 ist das dritte System der LEGO MINDSTORMS Reihe. Die Bezeichnung setzt sich aus EV für Evolution und 3 für die dritte Stufe der LEGO MINDSTORMS-Serie zusammen.<sup>25</sup>

Im Vergleich zu den Vorgängersystemen verfügt das EV3-System über eine modernere und leistungsfähigere Steuereinheit und auch die anderen elektronischen Komponenten des System wurden an den heutigen Stand der Technik angepasst.<sup>26</sup>

Die folgende Abbildung (68) zeigt einige der zentralen Komponenten des EV3-Systems, wie die Steuereinheit (EV3-Stein), Motoren und vier Sensoren.

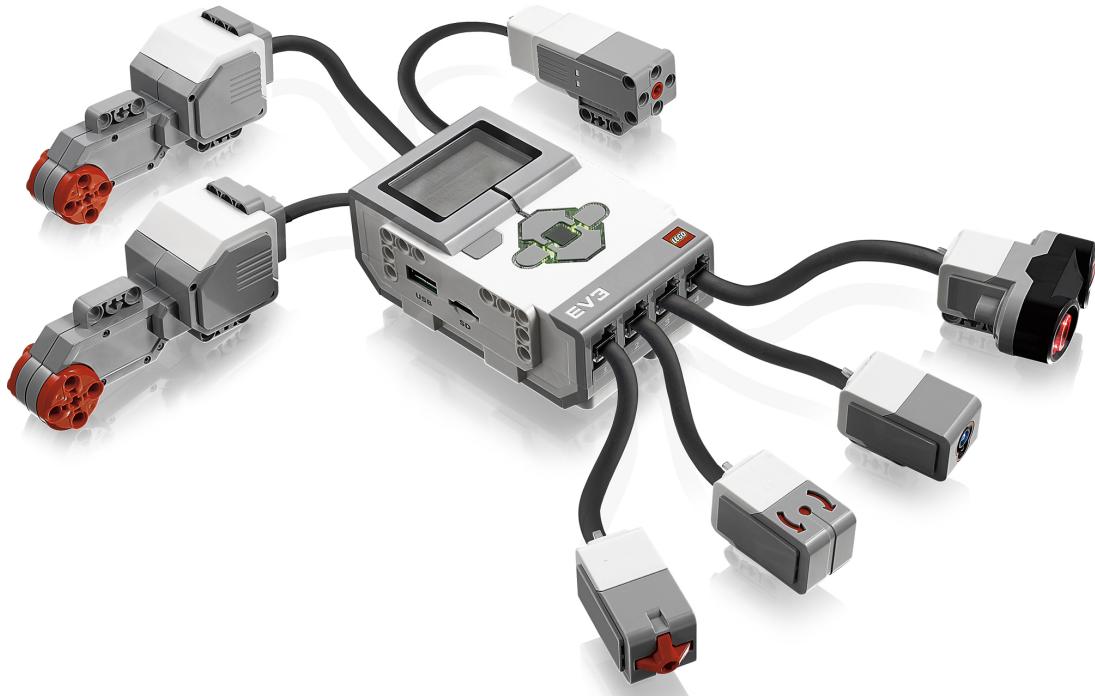


Abbildung 7: Zentrale Komponenten des EV3-Systems

Neben den elektronischen Komponenten gehören auch nicht elektronische Teile, wie Verbindungsstücke, Balken und Zahnräder, wie sie aus gängigen LEGO Produkten bekannt

<sup>25</sup>[vgl. ?, Das EV3 Roboter Universum, Seite 21]

<sup>26</sup>[vgl. ?, Das EV3 Roboter Universum, Seite 22]

---

sind, zum EV3-System. Sie bilden die strukturelle und mechanische Grundlage der Roboter.

Im Folgenden wird auf die elektronischen Komponenten des EV3-Systems näher eingegangen, da diese im Projekt eine deutlich größere Relevanz aufweisen.

### 2.2.2 Der EV3-Stein (Steuereinheit)

Die zentrale Komponenten und das Gehirn des LEGO MINDSTORMS EV3-Systems ist die zentrale Steuereinheit, kurz (EV3)-Stein oder auch Brick genannt. Bei ihm handelt es sich um einen Computer, welcher selbstständig Programme ausführen kann. Dazu verfügt der EV3-Stein über ein Linux Betriebssystem und eine spezielle Firmware, die wie die auszuführenden Programme auf einem Flash-Speicher liegen.<sup>27</sup>

Zur Kommunikation mit dem PC verfügt der EV3-Stein über eine USB- sowie Bluetooth-Schnittstelle. Neben der Kommunikation zu einem Computer kann die USB-Schnittstelle auch für den Zusammenschluss mit einem weiteren EV3-Stein (genannt Daisy Chain) genutzt werden.<sup>27</sup>

Für den Anschluss von Motoren und Sensoren verfügt der EV3-Stein über 8 Ports, an welche die anderen System-Komponenten über Kabel mit RJ12-Steckern angeschlossen werden. 4 der Ports dienen für den Anschluss von Motoren, die restlichen 4 Ports für die Abfrage von Sensorwerten.<sup>27</sup>

Der EV3-Stein besitzt an der Vorderseite ein LCD-Display zur Anzeige von Texten und Grafiken sowie 6 Knöpfe für die Bedienung durch den Benutzer. Display und Knöpfe dienen zur Bedienung der Firmware sowie zur Tätigung von Einstellungen, können aber ebenso durch Programme angesprochen und ausgewertet werden.<sup>27</sup>

Die folgende Auflistung zeigt einige Leistungsmerkmale des EV3-Steins.<sup>28</sup>

- Prozessor: ARM9 32Bit, 300 MHz, 16 MB Flash 64MB RAM
- Betriebssystem: Linux
- Sensoranschlüsse: 4x, Analog / Digital bis zu 460,8 Kbit/s
- USB-Schnittstellen: 2x, für Kommunikation zum PC, Daisy Chain, WiFi-Stick, USB-Speichermedium
- SD-Karten-Lesegerät: 1x, für MicroSD-Karte bis 32 GB
- User-Interface: 6 Knöpfe inkl. Beleuchtung
- Display: LCD Matrix, monochrom, 178 x 128 Pixel

---

<sup>27</sup>[vgl. ? , Das EV3 Roboter Universum, Seite 21]

<sup>28</sup>[vgl. ? ?, Das EV3 Roboter Universum, Seite 23 f., EV3-Programmieren mit Java, Seite 32]

- Kommunikation: Bluetooth v2.1, USB 2.0 (Kommunikation zum PC), USB 1.1 (Daisy Chain)

### 2.2.3 Motoren

Das EV3-System verfügt über zwei unterschiedliche Motoren, einen großen Motor und einen mittleren Motor. Bei beiden handelt es sich um Servomotoren mit integriertem Rotationssensor, welche von außen angesteuert und abgefragt werden können.<sup>29</sup> Die Motoren lassen sich sehr exakt steuern und ermöglichen so einen synchronen Betrieb mehrerer Motoren.<sup>30</sup>

Die folgende Tabelle zeigt die wichtigsten Eigenschaften der beiden Motoren.

Eigenschaft / Motortyp	Großer Motor	Mittlerer Motor
Winkelgenauigkeit	1 °	1 °
Umdrehungen	160 bis 170 U/min	240 bis 250 U/min
Drehmoment Rotation	20 Ncm	8 Ncm
Drehmoment Stillstand	40 Ncm	12 Ncm
Gewicht	76g	36g

Tabelle 2: Eigenschaften der EV3-Motoren

### 2.2.4 Sensoren

Zum EV3-System gehören eine Reihe von verschiedenen Sensoren, die es den Robotern ermöglichen Informationen über ihre Umwelt zu sammeln sowie ihre Eigenbewegungen zu erfassen. Im folgenden Abschnitt werden die wichtigsten Sensoren mit ihren Leistungsmerkmalen beschrieben.

**Farbsensor** Der Farbsensor ist ein digitaler Sensor, der dazu dient die Lichtintensität sowie verschiedener Farben zu erkennen. Der Sensor kann sowohl aktiv, als auch passiv betrieben werden und verfügt dafür über vier unterschiedliche Betriebsmodi:<sup>31</sup>

- Farbmodus (passiv) - In diesem Modus erkennt der Sensor 7 verschiedenen Farben.
- RGB-Modus (aktiv) - In diesem Modus sendet der Sensor nacheinander rotes, grünes und Balles Licht aus, je nachdem zu welchem Anteil ein Gegenstand die einzelnen Farben reflektiert wird die Farbe des Gegenstands ermittelt.
- Rotlicht-Modus (aktiv) - Bei diesem Modus wird Rotlicht ausgesendet und die Intensität des reflektierten Lichts gemessen.

<sup>29</sup>[vgl. ? , EV3-Programmieren mit Java, Seite 92]

<sup>30</sup>[vgl. ? , Das EV3 Roboter Universum, Seite 29 f.]

<sup>31</sup>[vgl. ? , EV3-Programmieren mit Java, Seite 101]

- Umgebungslicht-Modus (passiv) - Bei diesem Modus wird die Intensität des in das Sensorfenster eindringende Umgebungslichts gemessen.

Eigenschaften:

- Erkennung der Farben: keine Farbe, Schwarz, Blau, Grün, Gelb, Rot, Weiß, Braun
- Abtastrate: 1.000 Hz
- Entfernung: 15 bis 50 mm

Durch diesen Sensor wird es beispielsweise möglich, den Roboter einer farbigen Linie auf dem Boden folgen zu lassen.

**Ultraschallsensor** Diese aktive Sensor verwendet einen für den Menschen unhörbaren Ultraschall, um die Entfernung von Objekten zu ermitteln. Der Sensor emittiert dazu Ultraschall und misst die Laufzeit der Schallwellen, wenn diese von einem Objekt reflektiert werden, aus der Laufzeit kann dann die Entfernung ermittelt werden. Der Sensor verfügt über zwei unterschiedliche Betriebsmodi:<sup>32</sup>

- Messen - In diesem Modus sendet der Sensor Ultraschall aus, um die Entfernung von Objekten zu ermitteln.
- Scannen - In diesem passiven Modus emittiert der Sensor selbst keinen Ultraschall, sondern er reagiert auf fremden Ultraschall und kann so einen anderen aktiven Ultraschallsensor erkennen.

Eigenschaften:

- Genauigkeit: +/- 1 cm
- Messbereich: 3 cm bis 250 cm

**Berührungssensor** Der Berührungssensor ist ein einfacher mechanischer Sensor. Wird der Knopf am Ende des Sensors gedrückt wird dies registriert. Trotz der Einfachheit dieses Sensors ist dieser dennoch sehr nützlich, da er beispielsweise die Kollision des Roboters mit einem Hindernis erkennen kann.<sup>33</sup>

---

<sup>32</sup>[vgl. ? , Das EV3 Roboter Universum, Seite 32 f.]

<sup>33</sup>[vgl. ? , Das EV3 Roboter Universum, Seite 33]

---

**Kreiselsensor (Gyroskop)** Der Kreiselsensor ermöglicht es Drehbewegungen um eine Achse über Rotationsgeschwindigkeit und Drehwinkel zu messen. Dadurch wird es möglich die Eigenbewegung des Roboters oder einer Roboterkomponente zu registrieren.<sup>34</sup>

Eigenschaften:

- Genauigkeit: +/- 3° (bei einer 90° Drehung)
- Geschwindigkeit: maximal 440 Grad/Sekunde
- Abtastrate: 1.000 Hz

**Rotationssensor (Integriert)** Wie bereits im Abschnitt (2.2.3) dargelegt verfügen die beiden Motortypen über integrierte Rotationssensoren, die es ermöglichen die Umdrehungen der Motoren auszulesen. Durch diese Sensoren können mittels Odometrie Rückschlüsse über die Bewegung bzw. Position des Roboters geschlossen werden.

Eigenschaften:

- Genauigkeit: 1°
- Umdrehungen: Motorabhängig

Neben den hier vorgestellten Sensoren existiert noch ein Infrarotsensor, welcher in Verbindung mit einer Infrarotfernsteuerung dazu dient einen EV3-Roboter fernzusteuern.

## 2.2.5 Programmierung

Für die Programmierung der LEGO MINDSTORMS Produkte gibt es eine Reihe unterschiedlicher Programmiersprachen und -umgebungen. Die hauseigene LEGO-Software zur Programmierung des EV3 richtet sich an Einsteiger. Sie ermöglicht es über eine grafische Oberfläche via vorgefertigter Programmabläufe, welche durch grafische Blöcke repräsentiert werden den EV3 zu programmieren.<sup>35</sup>

Die Abbildung (3) gibt einen Überblick über verschiedene für den EV3 verfügbare Programmiersprachen sowie ihre Vor- und Nachteile.

---

<sup>34</sup>[vgl. ? , Das EV3 Roboter Universum, Seite 33]

<sup>35</sup>[vgl. ? , EV3-Programmieren mit Java, Seite 25 f.]

Eigenschaft / Programmiersprache	leJOS	EV3-Software	RobotC	NEPO
Installation	+	++	+	+++
Handhabung	+	++	+	++
Kosten	kostenlos	kostenlos	49\$	kostenlos
Einstieg	0	++	+	+++
Funktionsumfang	++	+	++	++

0 = neutral; + = gut; ++ = sehr gut; +++ = hervorragend

Tabelle 3: Eigenschaften der EV3-Motoren

**leJOS** Das LEGO Java Operating System abgekürzt leJOS ist ein Framework, dass es ermöglicht den EV3 mit der Programmiersprache Java zu programmieren. Das leJOS-Projekt wurde 1999 gegründet und sämtliche Komponenten (wie auch Java) sind kostenlos verfügbar.<sup>36</sup>

leJOS bietet eine schlanke **Java Virtual Machine (JVM)** für den EV3-Stein sowie eine Klassenbibliothek mit welcher die Komponenten des EV3 (Motoren, Sensoren etc.) angesprochen werden können. Installiert wird leJOS auf einer bootbaren microSD-Karte und kann anschließend davon gestartet werden, ohne die auf dem EV3 vorhandene LEGO-Software zu löschen oder zu verändern.<sup>37</sup>

Durch leJOS ist es möglich den EV3 mit Hilfe der Hochsprache Java zu programmieren, womit eine mächtige Programmiersprache zur Verfügung steht und die Vorteile der Objektorientierung für den EV3 genutzt werden können. leJOS bietet eine umfangreiche Klassenbibliothek sowie gut dokumentierte **Application Programming Interface (API)** was unter anderem die Integration von weiteren Sensoren etc. erleichtert.<sup>37</sup> Im folgenden sind einige Features die leJOS bietet aufgelistet:

- Objektorientierte Programmierung mit Java
- Die meisten Klassen der Pakete `java.lang`, `java.util` und `java.io`
- Rekursion
- Synchronisation
- Multithreading
- Exceptions
- Vollständige Bluetooth Unterstützung
- Umfangreiche Klassenbibliothek zum Steuern und Auslesen der EV3-Komponenten
- High-Level-Robotik-Tasks (Navigation, Lokalisation etc.)

<sup>36</sup>[vgl. ?, EV3-Programmieren mit Java, Seite 21]

<sup>37</sup>[vgl. ?, EV3-Programmieren mit Java, Seite 23]

## 2.3 App Entwicklung

Eine **App** ist ein ausführbares Programm für mobile Geräte, wie Smartphones oder Tablets. Um eine **App** für ein mobiles Gerät zu entwickeln, müssen vor Start der Entwicklung Anforderungen definiert sein, damit die Software spezifisch angepasst werden kann. Je nach Art der Anforderungen die an das System gestellt werden, bestehen verschiedene Möglichkeiten der Entwicklung. Allgemein kennt die **App** Entwicklung drei verschiedene Arten, die Native-, Web- und Hybride-Entwicklung, siehe Abschnitt (2.3.1), (2.3.2) und (2.3.3). Dabei werden verschiedene **Frameworks** verwendet, um mit unterschiedlichen Programmiersprachen den Aufbau der Logik zu beschreiben. Eine **App** besteht immer aus zwei Teilen, dem **User Interface (UI)**, das meist mit einer **Extensible Markup Language (XML)** oder **Hypertext Markup Language (HTML)** und **Cascading Style Sheets (CSS)** ähnlichen Sprache beschrieben wird und dem Programmcode, der sich auf viele Klassen verteilt und die Funktionalitäten der **App** beschreiben.



Abbildung 8: App-Entwicklung

[?]

### 2.3.1 Native Apps

In der Entwicklung von nativen **Apps** werden die direkten Ressourcen des Gerätes verwendet. Dazu gehört die Laufzeitumgebung des Betriebssystems, Bibliotheken und Hardwareschnittstellen. Der Vorteil von einer nativen Entwicklung liegt hauptsächlich darin, dass diese für das Betriebssystem optimiert ist und die vorhandenen Schnittstellen genutzt werden können, um komplexe und rechenintensive Anwendungen zu ermöglichen.<sup>38</sup> Vertreter dieser Entwicklung finden sich für verschiedene Betriebssysteme. Der populärste unter ihnen ist bei weitem Android mit einer nativen Java Entwicklung über Android Studio von Google. Native **Apps** besitzt aktuell den höchsten Marktanteil und eine entsprechende Popularität unter Entwicklern und Nutzern.

### 2.3.2 Web Apps

Die Entwicklung von web **Apps** arbeitet mit systemübergreifenden Ressourcen und greift auf gängige Webtechnologien, wie **HTML**, **CSS** und **JavaScript** zurück. Die **App** wird hierbei nicht wie normale Anwendungen direkt auf dem System des Gerätes ausgeführt, sondern kommt in dessen Browser zur Ausführung. Der Vorteil hierbei ist vor allem, dass diese Art von **App** auf allen Betriebssystemen lauffähig ist und direkt über das Internet

<sup>38</sup>[vgl. ?, Unterschiede und Vergleich native Apps vs. Web Apps]

---

veröffentlicht und aktualisiert werden kann, jedoch wird eine stabile Internetverbindung vorausgesetzt.<sup>38</sup>

Diese Entwicklung besitzt viele Vertreter mit der Unterstützung diverser **Frameworks**. Das populärste unter ihnen ist aktuell AngularJS von Google, was auf **JavaScript** basiert. In Kombination mit anderen Webtechnologien, wie **HTML** und **CSS** lassen sich performante **Web-Apps** entwickeln.

### 2.3.3 Hybride **Apps**

Die Entwicklung von hybriden **Apps** vereinigt die native- und webbasierte Entwicklung. Sie besteht dabei aus einem nativen Rahmen, in der eine **Web-App** zur Ausführung kommt, diese besitzt entsprechende Zugriffsrechte auf Hardwareschnittstellen, um diese mit **APIs** anzusprechen.<sup>39</sup>

Diese Entwicklung ist aktuell noch sehr jung, jedoch treten hier bereits verschiedene Vertreter hervor. Der populärste unter ihnen ist Ionic von Drifty, welches auf Apache Cordova als Basis zurückgreift. In Kombination mit AngularJS, **TypeScript** und anderen Webtechnologien lässt sich die hybride **App** entwickeln und auf einem beliebigen Gerät unter einem nativen Browser ausführen. Hybride **Apps** unterstützen dabei verschiedene Betriebssysteme, wie Android, iOS und Windows. Diese **Apps** können dabei meist nicht nur mobil, sondern unter anderem auf weiteren Systemen, wie stationäre zum Beispiel Desktop Rechner bereitgestellt werden.

### 2.3.4 Plattformübergreifende **Apps**

Um die Entwicklung von **Apps** einfach zu gestalten, verwenden immer mehr Entwickler die Form der plattformübergreifenden Entwicklung. Dadurch lässt sich die **App** unabhängig des Betriebssystem entwickeln und kann somit eine größere Menge von Nutzern erreichen. Diese Entwicklung greift dabei meist auf plattformübergreifende Konzepte, wie eine native Laufzeitumgebung oder einen nativen Browser zurück, um darin die **App** auszuführen. Der große Vorteil dieser **Apps**, liegt in der Wiederverwendbarkeit des Quellcodes und der verbesserten Wartbarkeit, da hier lediglich ein Projekt gewartet werden muss und der Quellcode für viele Betriebssysteme übernommen werden kann. Zur plattformübergreifenden Entwicklung wurden in den letzten Jahren viele Ansätze mit verschiedenen **Frameworks** entwickelt. Beispiele hierfür sind Ionic, Unity, Qt oder Xamarin.

---

<sup>39</sup>[vgl. ?, Native App, Web App und Hybrid App im Überblick]

### 2.3.5 Xamarin

Xamarin ist ein **Framework** zur Entwicklung von nativen, plattformübergreifenden **Apps**. Es basiert auf dem Mono Projekt, siehe Abschnitt (2.3.6), um damit auf verschiedenen Betriebssystemen, wie Android, iOS, Windows und Windows Phone ausgeführt werden zu können. Um nativen Quellcode auf den verschiedenen Systemen auszuführen, setzt Xamarin auf verschiedene Softwarekomponenten, um aus einem mit .NET entwickelten Projekt nativen Quellcode zu erzeugen.

Für iOS Systeme verwendet Xamarin den **Ahead of Time (AOT)** Compiler, um aus einem Xamarin.iOS Projekt, **Acorn RISC Machines (ARM)** Maschinencode zur erzeugen, der entsprechend schnell auf dem System ausgeführt werden kann.<sup>40</sup> Bei Android hingegen wird der Quellcode in **Intermediate Language (IL)** übersetzt, eine plattformübergreifende Assemblersprache, die durch das .NET **Framework** zur Ausführung gebracht wird. Die Übersetzung in **IL** geschieht mittels **Just-in-Time (JIT)**, um zur Laufzeit Maschinencode für das entsprechende Gerät zu erzeugen.<sup>40</sup> Damit dies bewerkstelligt werden kann, nutzt Xamarin zur Laufzeit Softwarekomponenten, die bestimmte Prozesse, wie Speicherverwaltung und Plattformoperationen verwalten.

Für eine effiziente Entwicklung bringt Xamarin eine große Bandbreite von Funktionalitäten für einen Entwickler. Beispiele sind dafür Bibliotheken, eine Test Cloud, sowie eine Unterstützung von nativen Bibliotheken, beispielsweise für **Java** oder **Objective-C**. Um mit Xamarin zu entwickeln, gibt es aktuell verschiedene Möglichkeiten auf unterschiedlichen Betriebssystemen. Einerseits kann mit Xamarin Studio auf einem OSX-System, oder mit Visual Studio auf Windows und Linux entwickelt werden.

Wie viele andere **Frameworks**, bietet auch Xamarin für verschiedene Zwecke nützliche **Templates**, die jeweils andere Nutzen besitzen. Die Entwicklung der **App** baut dabei vor allem auf zwei Hauptkomponenten von Bibliotheken, den Shared Projects und Portable Class Libraries.



Abbildung 9: Xamarin

[? ]

<sup>40</sup>[vgl. ?, Introduction to Mobile Development - Xamarin]

**Shared Projects** ermöglichen dem Entwickler Quellcode für verschiedene Plattformen zu entwickeln, wobei die plattformspezifischen Projekte das entsprechende Shared Project referenzieren. Somit besitzt diese Projektart keinen direkten Output, sondern kopiert den Quellcode entsprechend in das zu bauende Projekt, siehe Abbildung (10).<sup>41</sup> Der Hauptunterschied zu Standardprojekten liegt vor allem darin, dass ein Shared Project keine Abhängigkeiten haben darf und daher lediglich als Referenz für andere Projekte dienen kann.

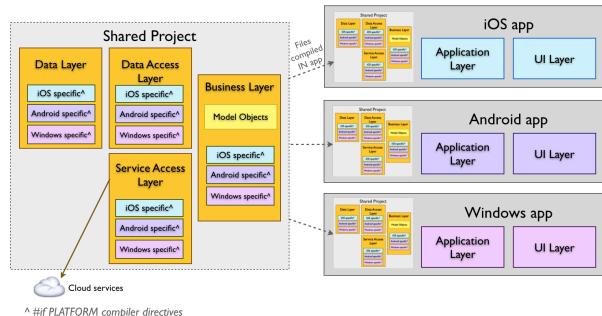


Abbildung 10: Shared Project

[?]

**Portable Class Libraries** ermöglichen dem Entwickler die Implementierung von plattformübergreifenden Bibliotheken, aus denen **Dynamic Linked Librarys (DLLs)** erzeugt werden können. Das Besondere an Portable Class Libraries ist dabei, dass die Plattformen spezifisch ausgewählt werden können, wobei auf die Unterstützung verschiedener Betriebssysteme zu achten ist, siehe Abbildung (12).<sup>42</sup> Eine Portable Class Library besitzt darüber hinaus verschiedene Vor- bzw. Nachteile, die für oder gegen ihre Nutzung sprechen.

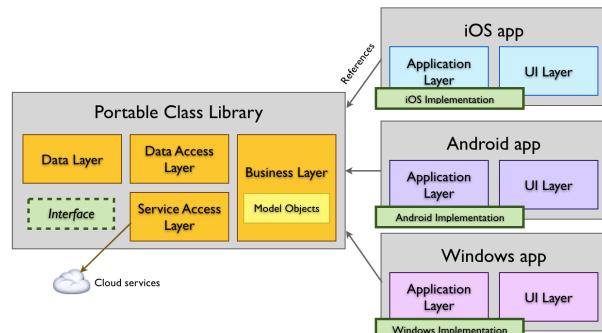


Abbildung 11: Portable Class Library

[?]

<sup>41</sup>[vgl. ?, Shared Projects - Xamarin]

<sup>42</sup>[vgl. ?, Introduction to Portable Class Libraries - Xamarin]

Feature	.NET Framework	Windows Store Apps	Silverlight	Windows Phone	Xamarin
Core	Y	Y	Y	Y	Y
LINQ	Y	Y	Y	Y	Y
IQueryable	Y	Y	Y	7.5 +	Y
Serialization	Y	Y	Y	Y	Y
Data Annotations	4.0.3 +	Y	Y		Y

Abbildung 12: Unterstützung Portable Class Library

[?]

Vorteile:

- Implementierung von zentralem Quellcode
- Einfaches Refactoring
- Referenzierung von Anwendungen

Nachteile:

- Keine Referenzierung von plattformspezifischen Quellcode
- Keine Standardbibliotheken vorhanden

**Xamarin.Forms** ist ein plattformübergreifendes UI Werkzeug zur Implementierung eines nativen Graphical User Interface (GUI) durch Extensible Application Markup Language (XAML), siehe Abbildung (13). Dieses basiert auf den typischen XML Strukturrichtlinien, womit sich die verschiedenen Elemente als klaren, übersichtlichen Text darstellen lassen, siehe Abbildung (14). Zur Strukturierung des Designs bietet Xamarin verschiedene Arten von Seiten-, Layout- und Navigationsprinzipien, welche durch Xamarin.Forms auf die nativen Elemente zurückgreifen. Damit lässt sich durch Xamarin.Forms native Designs schaffen, welche plattformübergreifend in Android, iOS und Windows angewendet werden können. Die vorhandenen Designs stellen dabei Klassen dar, die vom Entwickler entsprechend verwendet, oder bei Bedarf durch eigene Implementierungen angepasst werden können.

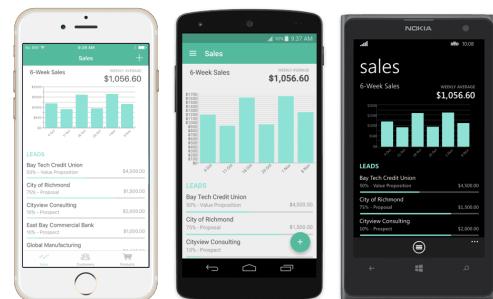


Abbildung 13: Design Xamarin.Forms

[?]

```
<?xml version="1.0" encoding="utf-8" ?>
<!ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FileAndCatch_App.Pages.SignInPage"
    Title="FileAndCatch">

    <StackLayout Orientation="Vertical">
        <StackLayout Orientation="Vertical" VerticalOptions="CenterAndExpand" HorizontalOptions="CenterAndExpand">
            <StackLayout Padding="20" HorizontalOptions="Center" VerticalOptions="Center">
                <Image Source="FileAndCatch_Logo.png" HeightRequest="200" WidthRequest="200"/>
            </StackLayout>
            <StackLayout WidthRequest="250" Orientation="Vertical" HorizontalOptions="Center" VerticalOptions="Center">
                <Entry Text="{Binding Connection.Address, Mode=TwoWay}" Placeholder="IP address" VerticalOptions="Center"/>
            <StackLayout Orientation="Horizontal">
                <Label Text="Save:" Margin="0, 0, 5, 5" VerticalOptions="Center" HorizontalOptions="End"/>
                <Switch IsToggled="{Binding Connection.Save, Mode=TwoWay}" VerticalOptions="Center" HorizontalOptions="Start"/>
            </StackLayout>
        </StackLayout>
        <Button Text="Sign in" FontSize="20" FontAttributes="Bold" TextColor="white" BackgroundColor="#0088BB" HeightRequest="50"
            VerticalOptions="End" Command="{Binding BSigin_OnCommand}"/>
    </StackLayout>
</ContentPage>
```

Abbildung 14: SignInPage in **XAML**

Xamarin.Forms setzt mit seinem Layout auf bekannte Designprinzipien, welche bereits in anderen Frameworks, wie .NET vorzufinden sind, siehe Abbildung (15) und (16). Diese können in drei Kategorien eingeteilt werden, einerseits die eigentliche Seite, die in verschiedene **UIs** dargestellt werden können und somit die Nutzerinteraktion beeinflussen. Andererseits das Layout, welches die Darstellung der vorhandenen Elemente regelt und den Pages, die für die Navigation unter den einzelnen Seiten zuständig sind. Dem Entwickler sind dabei keinerlei Grenzen gesetzt, wobei dieser sämtliche zur Verfügung stehenden Ressourcen miteinander kombinieren kann.

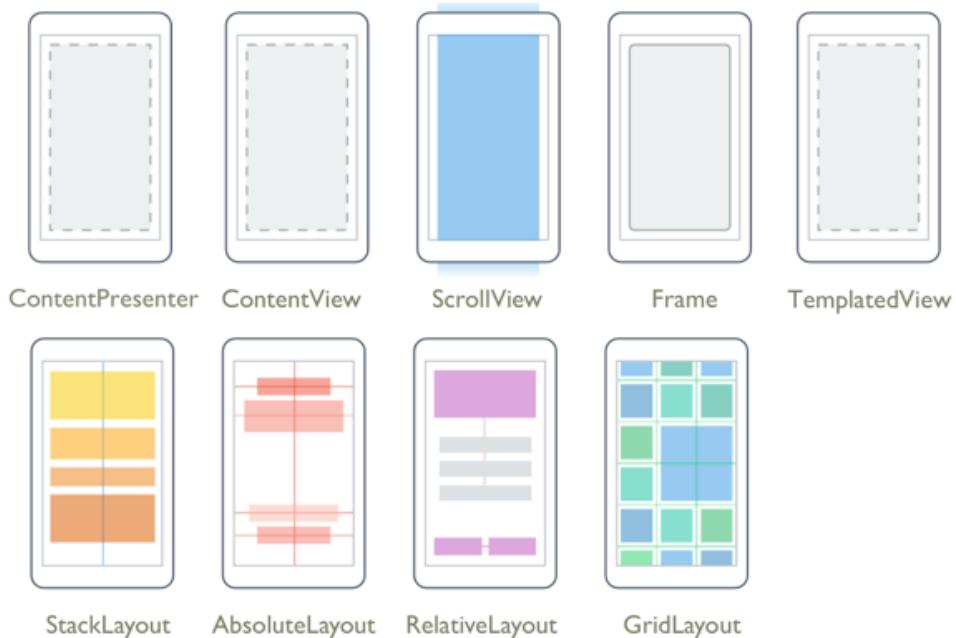


Abbildung 15: Xamarin.Forms Layouts

[? ]



Abbildung 16: Xamarin.Forms Pages

[?]

### 2.3.6 Mono

Mono ist ein Open-Source-Framework, das auf dem .NET Framework von Microsoft basiert. Die Implementierung von Mono greift dabei auf die Standards von .NET für die Programmiersprache **C#**, sowie die **Common Language Infrastructure (CLI)** zurück.<sup>43</sup> Dies ermöglicht Entwicklern die Erstellung von plattformübergreifenden Anwendungen, welche mittels einer zur Verfügung gestellten Laufzeitumgebung auf verschiedenen Systemen ausgeführt werden können.

Um Anwendungen auf verschiedenen Systemen auszuführen, nutzt Mono verschiedene Komponenten. Dazu gehört an vorderster Stelle ein Compiler, um den erstellten Quellcode in die jeweilige Maschinensprache zu übersetzen. Die Übersetzung findet dabei in Kooperation mit der Mono Runtime statt, welche die entsprechende Infrastruktur zur Ausführung der Anwendung bereitstellt. Für eine effiziente Entwicklung stellt Mono zwei Bibliotheken zur Verfügung, einerseits die .NET Class Library, die die Grundelemente von .NET enthält, sowie die Mono Class Library mit zusätzlichen Funktionen für plattformübergreifende Anwendungen.

Im Vergleich mit anderen Frameworks sprechen verschiedenen Vorteile für die Nutzung von Mono. Der Hauptgrund für die Nutzung liegt vor allem in der Popularität von .NET, da dies auf den meisten Rechnern zur Verfügung steht, oder installiert werden kann. Ein großer Nutzen stellt die High-Level-Programmierung dar, welche eine Implementierung mit einer Laufzeitumgebung ermöglicht, die Funktionen wie Speicherverwaltung selbst organisiert. Durch Verwendung der **Common Language Runtime (CLR)** kann der Entwickler seine übliche Programmiersprache verwenden und ist unabhängig vom bestehenden System.



Abbildung 17: Mono

[?]

<sup>43</sup>[vgl. ?, About Mono]

### 2.3.7 .NET Framework

Das .NET Framework dient zur Entwicklung sowie Ausführung von Anwendungen, die mit Programmiersprachen implementiert sind, welche auf den Standards von .NET basieren. Es besteht aus verschiedenen Komponenten, wobei der Kern des Frameworks in der **CLR** liegt.<sup>44</sup> Diese ist verantwortlich für die Laufzeitumgebung und somit für die Ausführung der Anwendungen, indem es die bereitgestellten Ressourcen des Systems nutzt.

---

<sup>44</sup>[vgl. ?, Overview of the .NET Framework]

Die **CLR** führt zur Laufzeit, je nach System, verschiedene Aktionen aus, um die entsprechende Anwendung auszuführen. Der allgemeine Ablauf ist dabei folgender: Der Quellcode wird in die **CLR** geladen und nach entsprechenden Sicherheitsanforderungen des Systems überprüft.<sup>44</sup> Anschließend wird er durch eine **JIT** Kompilierung in einen **IL**-Quellcode konvertiert, um diesen nativ auf dem System ausführen zu können.<sup>44</sup> Der **IL**-Quellcode setzt dabei auf die gesetzten Standards der **CLI** auf, die eine sprach- und plattformunabhängige Entwicklung von Anwendungen ermöglicht.<sup>44</sup>

Das .NET Framework bietet zusätzlich zur

unabhängigen Entwicklung verschiedene unterstützende Komponenten. Die wichtigste unter ihnen ist die .NET Class Library. Diese unterstützt den Entwickler mit einer Sammlung bereits implementierten Quellcodes, wie Klassen und entsprechenden Zugang zu systemnahen Schnittstellen. Mit dem .NET Framework lässt sich eine große Bandbreite von Anwendungen entwickeln, von Konsolenanwendungen, über grafischen Oberflächen, bis hin zu Webanwendungen.

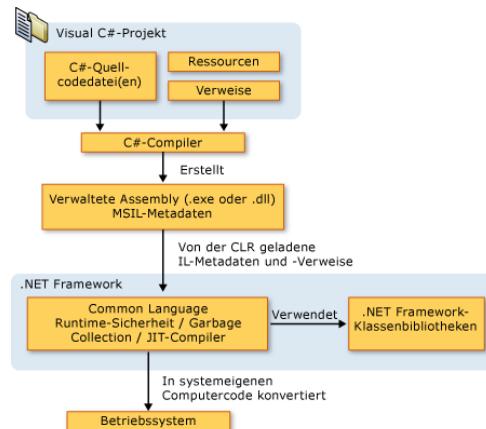


Abbildung 18: .NET Framework Ausführung

[?]

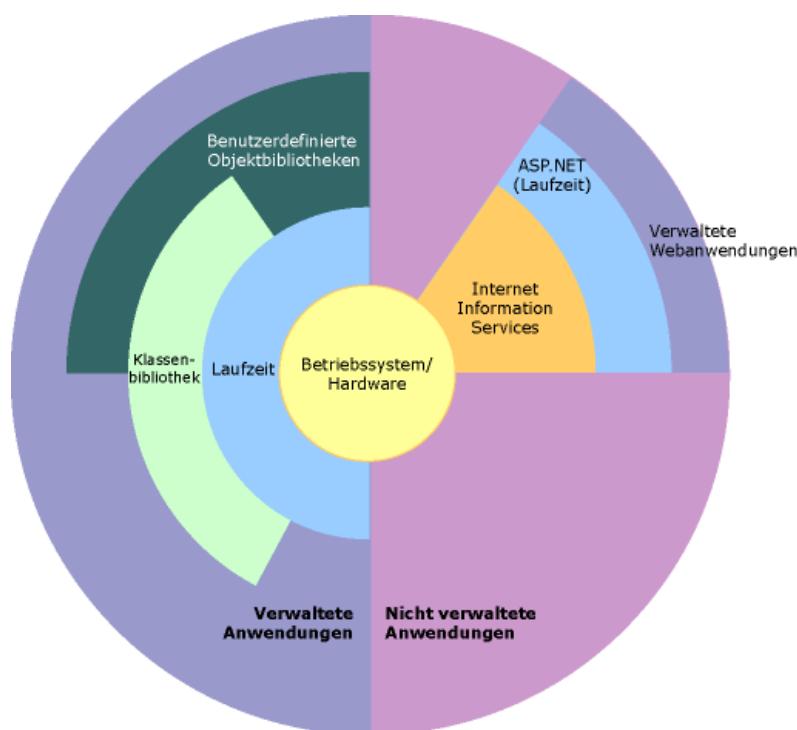


Abbildung 19: Common Language Runtime

[?]

## 2.4 TCP-Kommunikation

Das **Transmission Control Protocol (TCP)** ist ein Transportprotokoll und ermöglicht eine Datenaustausch zwischen kommunizierenden Anwendungsinstanzen in einer Ende-zu-Ende-Beziehung. **TCP** ist als Transportprotokoll in der vierten Schicht des **Open Systems Interconnection (OSI)**-Modells angesiedelt und basiert auf dem **IP** mit dem es zusammen als Namensgeber der TCP/IP-Protokollfamilie (Internetprotokollfamilie) dient. **TCP** ist ein offenes, frei verfügbares und weit verbreitetes Protokoll. Als Mitglied der Internetprotokollfamilie ist **TCP** neben **Unified Datagram Protocol (UDP)** das Transportprotokoll, auf dem die meisten Anwendungen im Internet basieren.<sup>45</sup>

### 2.4.1 Gundlegendes

Als verbindungsorientiertes Protokoll sorgt TCP für die Erzeugung und Erhaltung einer gesicherten Ende-zu-Ende-Verbindung zwischen zwei Anwendungsprozessen. TCP arbeitet paketvermittelt d.h. überträgt Daten Paketweise und ist ein zuverlässiges Protokoll. Durch diese Eigenschaften stellt TCP sicher, dass Daten

- nicht verloren gehen
- nicht verändert werden
- nicht dupliziert werden
- in der richtigen Reihenfolge eintreffen

Zur Gewährleistung einer vollständigen Übertragung sowie der Integrität der gesendeten Daten nutzt TCP Prüfsummen, Bestätigungen, Zeitüberwachungs- und Nachrichtenwiederholungsmechanismen sowie Sequenznummern für die Reihenfolgeüberwachung und das Sliding Windows Prinzip zur Flusskontrolle.<sup>46</sup>

TCP nutzt prinzipiell folgende Protokollmechanismen:

- Drei-Wege-Handshake-Verbindungsauflauf- und -abbau
- Positives, kumulatives Bestätigungsverfahren mit Timerüberwachung für jede Nachricht
- Implizites negatives Bestätigungsverfahren (NAK-Mechanismus): Bei drei ankommenen Duplikat-ACK-PDUs wird beim Sender das Fehlen des folgenden Segments angenommen. Ein sog. Fast-Retransmit-Mechanismus führt zur Neuübertragung des Segments, bevor der Timer abläuft.
- Pipelining

<sup>45</sup>[vgl. ? , Grundkurs Datenkommunikation, Seite 189]

<sup>46</sup>[vgl. ? , Grundkurs Datenkommunikation, Seite 190]

- Go-Back-N zur Übertragungswiederholung
- Fluss- und Staukontrolle

#### 2.4.2 Nagle-Algorithmus

Nagle-Algorithmus (RFC 896 und RFC 1122) ist ein Algorithmus, der der Optimierung dient und der bei allen **TCP**-Implementierungen verwendet wird. Der Nagle-Algorithmus versucht aus Optimierungsgründen zu verhindern, dass viele kleine Nachrichten gesendet werden, da dies schlecht für die Netzauslastung ist.<sup>47</sup>

Dazu werden mehrere Nachrichten zusammengefasst und gebündelt versendet, dies geschieht nach folgendem Prinzip:

- Erhält der **TCP**-Endpunkt Daten vom Anwendungsprozess wird zunächst nur das erste Datenpaket gesendet und die restlichen Daten werden im Sendepuffer gesammelt.
- Danach werden weitere Daten so lange im Sendepuffer gesammelt bis alle zuvor gesendeten Datenpakete vom Empfänger bestätigt wurden oder so viele Daten im Sendepuffer liegen, dass die eingestellte Segmentgröße erreicht ist und ein volles Datenpaket gesendet werden kann.

Dieses Verfahren sorgt zwar für eine gute Netzauslastung, da das Verhältnis von Nutzdaten zu Overhead (**TCP**-Header etc.) steigt, jedoch ist dies nicht für alle Anwendungsszenarien optimal, da es die Latenz erhöht. Insbesondere bei Anwendungen die eine unmittelbare Antwort der Gegenstelle benötigen wie **Secure Shell (SSH)**- oder Telnet-Anwendung sorgt dies für Verzögerungen. In diesem Fall ist es besser den Nagle-Algorithmus auszuschalten.<sup>48</sup>

#### 2.4.3 Kommunikationsablauf

Da es sich bei **TCP** um ein verbindungsorientiertes Protokoll handelt gliedert sich die Kommunikation in drei Phasen: Verbindungsaufbau, Datenaustausch und Verbindungsabbau. Bevor Daten übertragen werden können, muss die Verbindung durch den Verbindungsaufbau initiiert und nach Beendigung der Datenübertragung wieder abgebaut werden.

**Client & Server** Der Verbindungsaufbau einer Kommunikation erfolgt bei **TCP** nach dem Client-/Server-Paradigma, d.h. einer der Teilnehmern agiert als Server und wartet auf einen Verbindungsaufbau durch den Client, welchen der andere Teilnehmern darstellt.

---

<sup>47</sup>[vgl. ? , Grundkurs Datenkommunikation, Seite 198]

<sup>48</sup>[vgl. ? , Grundkurs Datenkommunikation, Seite 198 f.]

Nach dem Verbindungsauflauf haben die beiden Rollen jedoch keine Bedeutung mehr und die beide Teilnehmern sind sowohl bei der Datenübertragung, als auch beim Verbindungsabbau gleichberechtigt.

**Verbindungsauflauf** Der Verbindungsauflauf bei **TCP** basiert auf dem Three-Way-Handshake. Dabei schickt der Client einen Verbindungswunsch (SYN) an den Server. Der Server bestätigt den Erhalt der Nachricht (ACK) und äußert seinerseits einen Verbindungswunsch (SYN), welchen der Client nach Erhalt der Nachricht bestätigt (ACK). Nach Ablauf dieses gegenseitigen Anfrage- und Bestätigungsorgangs ist die Verbindung initiiert und der Datenaustausch zwischen den Teilnehmern kann beginnen.<sup>49</sup>



Abbildung 20: TCP Verbindungsauflauf  
[?]

**Datenaustausch** Zum Datenaustausch werden die Daten in einzelnen Datenpakete verpackt und an den Empfänger gesendet welcher den Erhalt jedes einzelnen Paketes bestätigen (ACK) muss. Für jedes gesendete Paket existiert beim Sender ein Timer. Ist dieser abgelaufen ohne das eine Bestätigung des Empfangs für das entsprechende so muss der Sender davon Ausgehen, dass das Paket verloren gegangen ist und Sendet dieses erneut. Durch dieses Verfahren ist gewährleistet das auch im Fall von Paketverlusten keine Daten verloren gehen. Da es sich bei **TCP** um eine bidirektionale Verbindung handelt kann nicht nur eine Seite Daten versenden sondern beide Teilnehmer können Daten senden und empfangen.<sup>50</sup>

**Verbindungsabbau** Nach Abschluss der Datenübertragung wird von einer Seite (egal von welcher) ein Verbindungsabbau initiiert. Dazu dient ein etwas modifizierter Dreie-Wege-Handshake-Mechanismus. Jede der beiden Verbindungsrichtungen der Vollduplex-Verbindung wird abgebaut, d.h. beide Seiten bauen ihre „Senderichtung“ ab. Die initiierende Seite schickt zuerst einen Verbindungsabbauwunsch (FIN). Die Gegenstelle bestätigt den Erhalt der Nachricht (ACK) und schickt ebenfalls einen Verbindungsabbauwunsch (FIN), woraufhin sie von der Gegenstelle noch mitgeteilt bekommt, dass die Verbindung abgebaut ist (ACK).

<sup>49</sup>[vgl. ?, TCP-Kommunikation]

<sup>50</sup>[vgl. ?, TCP-Kommunikation]

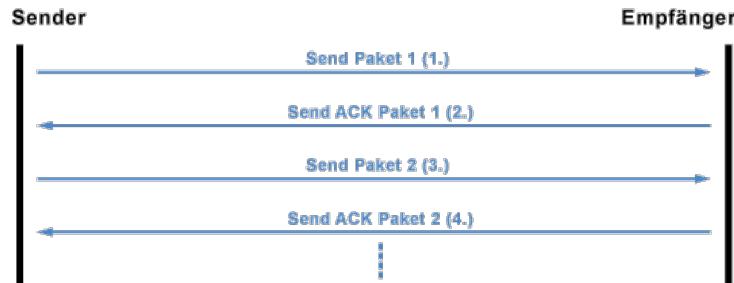


Abbildung 21: TCP Datenaustausch  
[?]

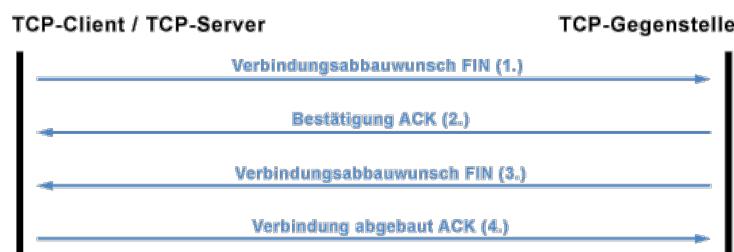


Abbildung 22: TCP Verbindungsabbau  
[?]

#### 2.4.4 Socket-Programmierung

Als Transportzugriffsschnittstelle für die **TCP**-basierte Kommunikation dient die Socket-Schnittstelle. Obwohl es sich bei **TCP** ein paketvermitteltes Protokoll handelt der Anwendung eine Strom-orientierte Kommunikation, die Daten werden also von einem Anwendungsprozess Byte für Byte in einem Bytestrom geschrieben und **TCP** sorgt anschließend um den Aufbau von Segmenten, die dann übertragen werden. Andere Transportdienste erwarten ihre Daten in festen Blöcken.<sup>51</sup>

<sup>51</sup>[vgl. ?, Grundkurs Datenkommunikation, Seite 96 f. f.]

## 2.5 Schwarmverhalten

Dieser Abschnitt beschäftigt sich mit der Theorie des Schwarmverhalten und dessen Abstammung.

### 2.5.1 Allgemein

Das Schwarmverhalten beschreibt die Verhaltensweise eines Schwarmes, welcher aus einer einheitlich formierten Tierart besteht. Diese interagieren untereinander, um eine evolutionstechnische Überlegenheit, durch das Bilden eines Schwarmes zu erhalten. Der Vorteil liegt dabei hauptsächlich im Schutz vor Fressfeinden des einzelnen Individuums, indem diese nicht als Beutetier identifiziert werden können, was als Konklusionseffekt bekannt ist sowie die Zusammenarbeit der einzelnen zur Jagd auf andere Tiere.<sup>52</sup> Zusätzlich bietet es die Möglichkeit sich gegenüber Fressfeinden zu behaupten und Jungtiere zu schützen. Zudem dient es einer Erleichterung der Fortpflanzung, da in einem Schwarm eine entsprechende Auswahl an paarungsbereiten Partnern vorhanden ist, um eine genetische Vielfalt zu erreichen.<sup>52</sup>

### 2.5.2 Vorbilder aus dem Tierreich

Zur Nachstellung eines Schwarmverhaltens existieren verschiedene Vorbilder, die aus dem Tierreich übernommen werden können. Dabei verwendet jede Tierart ihre eigenen Überlebensstrategien, sowie Regeln die in dem anzutreffenden Schwarm vorherrschen.

**Ein Fischschwarm** ist hierfür eines der bekanntesten Beispiele, indem die einzelnen Individuen des Schwarmes sich so verhalten, als würden sie wie ein einzelnes Tier agieren. Dabei setzt jeder Fisch feste Regeln um, die zur Umsetzung eines Schwarmverhaltens führen.

1. Folge deinem Vordermann
2. Behalte die Geschwindigkeit deines Nachbarn bei



Abbildung 23: Fischschwarm

Die wichtigste Rolle spielt hierbei der Schwellenwert zur Steuerung des Schwarms durch einzelne Teilnehmer. Da jedes Individuum den Schwarm durch seine Bewegung beeinflusst, existiert ein Schwellenwert, der sich einer Mindestanzahl von etwa fünf Pro-

<sup>52</sup>[vgl. ?, Schwarmverhalten]

zent der Fische richtet, durch die der Schwarm gesteuert werden kann.<sup>53</sup> Somit reagieren die Fische ausschließlich auf die Mehrheit und der Schwarm lässt sich nicht durch eine Minderheit Fehlsteuern. Dieses Szenario lässt im Sinne von Veranstaltungen auf Menschen übertragen, um zu festzustellen an welchen Positionen Sicherheitspersonal positioniert werden muss, um einen geregelten Ablauf zu gewährleisten.<sup>53</sup>

**Ein Termitenschwarm** geht nach dem Prinzip der Stigmergie vor, wie andere Insekten, die vorwiegend in einem Staat leben. Dabei kommunizieren die einzelnen Individuen indirekt über die Beeinflussung ihrer Umgebung, wie dem Hinterlassen von Spuren als Merkmale.<sup>54</sup> Dieses Prinzip befähigt den kleinsten und primitivsten Organismen einen evolutionären Vorteil zu erlangen, indem sie sich nicht allein den Gefahren stellen, sondern in einer großen Masse zusammenarbeiten.

Dies ermöglicht ihnen das Bauen riesiger Nester, indem die Insekten einen ihren vorgegebenen inneren Plan, mit ihrer aktuellen Umgebung vergleichen und dadurch intuitiv erkennen, welche Arbeit sie zu erledigen haben.<sup>54</sup>

Der Ablauf ist dabei folgender:

1. Erkennen
2. Analysieren
3. Reagieren

Dieses Prinzip lässt sich durch festlegen von Regeln auf Roboterschwärme abbilden, um wie Insekten vollkommen autonom Gebäude und andere Objekte durch den Einsatz einer indirekten Kommunikation zu errichten.

### 2.5.3 Szenarien

Zur Umsetzung verschiedener Schwarmverhalten lassen sich nützliche Teile der Tierwelt auf Roboter abstrahieren, um eine Zusammenarbeit der einzelnen Individuen effizienter zu gestalten. Hierbei werden meist mehrere Basisszenarien eingesetzt, die einerseits die Kommunikation, oder die Abläufe verbessern.

<sup>53</sup>[vgl. ?, Logistik Schwarmintelligenz]

<sup>54</sup>[vgl. ?, Roboter bauen im Schwarm nach dem Vorbild von Termiten]



Abbildung 24: Termitenschwarm

Für die Umsetzung der Kommunikation der einzelnen Teilnehmer existieren zwei verschiedene Möglichkeiten, einerseits eine direkt, oder indirekte Kommunikation. Dabei kann bei einer direkten Kommunikation auf die bekannten Kommunikationsmittel zurückgegriffen und über ein existierendes Netzwerk kommuniziert werden. Dabei wird meist nach dem bekannten Request-Response-Prinzip vorgegangen.

1. Anfrage
2. Verarbeitung
3. Antwort

Bei einer indirekten Kommunikation analysiert der Roboter dagegen seine Umgebung, um daraus Veränderungen zu registrieren und entsprechend zu reagieren. Dabei wird bei einer Umsetzung zur Implementierung folgend vorgegangen:

1. Datenerfassung durch Sensorik
2. Datenauswertung
3. Vergleich der Datensätze
4. Reaktion und Ausführung

Für die Umsetzung eines Schwarmverhaltens, können viele Szenarien herangezogen werden, je nachdem, welche Problemstellung gelöst werden soll. Da sich dieses Projekt auf mobile Kleinroboter bezieht kommen folgende Basisszenarien in Frage:

- Synchrone Verhaltensweise
- Verfolgung eines Objektes
- Selbstständige Verhaltensweise

**Die Synchrone Verhaltensweise** stellt einen Schwarm dar, wobei alle Individuen als Einheit agieren und somit ein großes Tier nachahmen. Dazu werden meist den jeweilig beteiligten Robotern dieselben Informationen zur Verfügung gestellt, um eine gleichmäßige Reaktion zu erreichen. Dies dient dazu spezielle Formen mit Robotern darzustellen und den gesamten Schwarm zu steuern.



Abbildung 25: Roboter bilden Formen  
[? ]

---

**Die Verfolgung eines Objektes** stellt Roboter dar, welche einem klar definierten Objekt folgen. Dies kann durch verschiedene Prinzipien veranlasst werden. Einerseits kann ein Objekt durch vorhandene Sensorik erfasst und damit verfolgt werden. Andererseits kann eine Verfolgung mittels Positionsdaten, wie **GPS** realisiert werden. Dieses Prinzip eines Schwarmverhaltens erfasst damit ein weites Spektrum an Anwendungsgebieten, wie eine automatisierte Verkehrsführung oder der Verfolgung durch Drohnen, welche die Möglichkeit besitzen durch Gesichtserkennung Personen und Objekte zu erkennen.



Abbildung 26: Roboter bilden Formen

[? ]

**Die Selbstständige Verhaltensweise** stellt eine indirekte Kommunikation zwischen Robotern dar, welche eine asynchrone Zusammenarbeit dieser ermöglicht. Dabei können unabhängig voneinander Roboter nach ihren Stärken für ein gemeinsames Ziel zusammenarbeiten. Dies findet oft Anwendung in Industrieanlagen, wobei jeder Roboter für eine spezielle Aufgabe ausgerüstet und programmiert ist. Diese erfassen zunächst das Produkt, um anschließend ihren definierten Arbeitsschritt durchzuführen.



Abbildung 27: Industrieroboter

[? ]

## 3 Konzeption

In diesem Kapitel werden die Anforderungsdefinitionen des Projektes, mit Spezialisierung auf die verschiedenen Use Cases beschrieben.

### 3.1 Anforderungsdefinitionen

Ein Schwarmverhalten zur Interaktion von Kleinroboter benötigt verschiedene Anforderungen, um korrekt untereinander agieren zu können. Die Basis hierbei, bildet das Kommunikationssystem zwischen den einzelnen Komponenten, um erfasste Daten zuverlässig zu synchronisieren. Zur Interpretation der entsprechenden Daten benötigt jede Komponente den jeweiligen Aufbau und die Struktur der Kommunikationsdaten, damit diese verwertet und Aktionen ausgeführt werden können.

Diese Aktionen repräsentieren den Grundbestandteil des Schwarmverhaltens und sind auf verschiedenen Systeme verteilt. Die Roboter benötigen hierbei implementierte Funktionen, wie das Ansteuern von Motoren, Sensorik, sowie die Aktualisierung, um erfasste Daten an Nutzer weiterzuleiten. Zur Steuerung dient eine **App** für mobile Smartphones mit einem **UI** um verschiedene Szenarien zu starten, sowie die Roboter kontrollieren zu können. Die Kontrollschnittstelle stellt dabei eine Desktopanwendung dar, über die der Nutzer mit den Robotern kommuniziert und Daten zur Steuerung abgreifen kann, wobei mehrere Nutzer zur selben Zeit mit verschiedenen Szenarien unterstützt werden sollen.

Damit bestehen folgende Anforderungsdefinitionen an die zu erstellenden Softwarekomponenten:

- Kommunikationssystem
- Interpretation
- **UI**
- Steuerungsfunktionen

### 3.2 Softwarearchitektur

Die Architektur der Software besteht aus drei Hauptkomponenten, den Robotern, einer Desktopanwendung, sowie einer mobilen App, siehe Abbildung (28).

Diese Komponenten kommunizieren über ein drahtloses Netzwerk mittels TCP untereinander, indem diese Zeichenketten als JSON versenden. Dadurch lassen sich gesammelte Daten als Objekte kapseln und auf den verschiedenen Systemen entsprechend synchronisieren. Dies geschieht über eine Klassenstruktur, die Kommandos abbildet, durch welche die kommunizierten Daten serialisiert und als Objekte dargestellt werden können, siehe Abschnitt (3.6).

Die Programmierung der Roboter basiert auf dem Betriebssystem leJOS, welches eine Java Virtual Machine (VM) auf dem EV3 zur Verfügung stellt. Durch die dadurch bereitgestellten Bibliotheken ist eine unkomplizierte Implementierung der Logik möglich sowie steht eine direkte Unterstützung in Eclipse zur Verfügung, welche ein Debuggen der Software gestattet. Die Roboter unterstützen für ein Schwarmverhalten klassische Funktionen, um die Daten der vorhandenen Sensorik auszulesen, sowie Motoren anzusteuern. Diese Funktionen werden einerseits durch Kommandos ausgeführt, um den entsprechenden Roboter zu steuern. Andererseits werden regelmäßig Daten durch einen Prozess erfasst, um diese auf dem Backend zu aktualisieren.

Die App beruht auf der plattformübergreifenden Implementierung mittels des Frameworks Xamarin um möglichst viele Systeme zu erreichen. Sie baut dabei auf ein einfaches UI mit dem Design Pattern Model-View-ViewModel (MVVM) auf, um diese von der eigentlichen Logik zu trennen und einen qualitativ hochwertigen Quellcode zu schaffen, der einfach gewartet werden kann. Die App besitzt Basisfunktionen zur Erstellung von Kommandos, die die Verwaltung von Szenarien ermöglicht und steuert somit den Schwarm.

Das Backend dient als zentrale Kommunikationsschnittstelle des gesamten Systems und steuert die Kommandos für den Ablauf der Szenarien. Es besitzt ein UI mit JavaFX Realisierung zur Anzeige von erfassten Daten und stellt diese anhand einer auswählbaren Hierarchie dar.



Abbildung 28: Softwarearchitektur

### 3.3 Steuerung

Die Steuerung des Roboterschwarms greift in sämtlichen implementierten Szenarien auf die Sensorik des Smartphones als Basis zurück. Verwendet werden hierbei die Bewegungssensoren, um eine Steuerung durch das Hin- und Herschwenken des Smartphones zu ermöglichen. Dies stellt eine intuitive Steuerung dar und ist für jeden neuen Nutzer schnell begreiflich. In Abbildung (29) ist die entsprechende Steuerung zur Bewegung des Roboters dargestellt. Um die Roboter möglichst genau zu steuern, erfasst die Sensorik laufend Daten, welche im **UI** angezeigt werden. Dadurch lässt sich eine Veränderung der Daten darstellen, die zu einer signifikant verbesserten Steuerung führen.



Abbildung 29: Steuerung

### 3.4 Szenarien

Zur Realisierung des Roboterschwarms greift die Software auf verschiedene definierte Szenarien als Kontext zurück. Diese sind in Control, Synchron, Follow, Flee und Catch untergliedert, wobei ein Single mit einem Nutzer oder einem Mehrnutzersystem als Multi unterschieden wird. Der Multi Mode dient hierbei als Erweiterung zur Software und ist im vorliegenden System nicht geplant und kann daher nicht genutzt werden. Folgend werden die einzelnen Szenarien beschrieben, die als Kontext für ein Schwarmverhalten genutzt werden können.

---

**Control** stellt eine direkte Steuerung eines einzelnen Roboters dar und fällt somit nicht unter die Kategorie Schwarmverhalten. Dieses Szenario dient zur Entwicklung der grundlegenden Funktionen, auf denen das Schwarmverhalten und damit weitere Szenarien aufbauen.

**Synchron** stellt eine synchrone Steuerung von mehreren Robotern dar, in dessen Kontext jeder beteiligte Roboter identische Kommandos erhält. Durch eine entsprechende Aufstellung der Roboter lassen sich Schwärme aus dem Tierreich, wie Fische oder Vögel nachahmen.

**Follow** stellt eine Reihe von Robotern dar, indem der vorderste vom Nutzer gesteuert werden kann. Die restlichen Roboter erhalten ihre Position in der Schlange entsprechend der Position ihres Vordermannes, dem sie stetig folgen. Da diese Ablauf laufend wiederholt wird, stellen alle Roboter gesamt eine Schlange dar, wobei die einzelnen die Muskeln und der vorderste Roboter den Kopf repräsentiert.

**Flee** stellt ein Verfolgungsszenario dar, indem der Nutzer mit seinem Roboter vor anderen flieht. Dabei erhalten die restlichen Roboter laufend eine Position um immer näher an diesen heranzufahren. Sollte der Nutzer durch einen Roboter gefangen werden, ertönt ein Endsignal, wobei anschließend das Szenario beendet wird.

**Catch** stellt ein Verfolgungsszenario dar, indem der Nutzer die restlichen Roboter fängt. Diese fahren zufällig in verschiedene Richtungen davon. Sollte der Nutzer alle gefangen haben, ertönt ein Signal und beendet damit das Szenario.



Abbildung 30: Logo

## 3.5 Use Cases

In diesem Abschnitt werden die Use Cases des Schwarmverhaltens beschrieben. Dabei wird insbesondere auf den Ablauf in Form von **Unified Modeling Language (UML)** Diagrammen eingegangen.

### 3.5.1 Connection

Der Use Case Connection stellt den Ablauf eines Verbindungsaufbaus zwischen den Komponenten und der Desktopanwendung dar. Dies soll über die Nutzung eines drahtlosen Netzwerks mittels **TCP** Schnittstelle des Smartphones realisiert werden. Die IP-Adresse kann dabei durch die Verwendung einer **SQLite** Datenbank lokal gespeichert werden, um diese später bei einem erneuten Aufruf automatisch eintragen zu lassen. Zur Unterscheidung der verschiedenen Komponenten soll eine individuelle Identifikation erstellt werden, wobei der Typ der Komponente, sowie weitere Merkmale ersichtlich werden sollen.

Der Use Case soll dabei in zwei unterschiedliche Typen untergliedert werden, womit ein Verbindungsauflauf von einem Verbindungsabbau unterschieden werden kann. Der Ablauf eines Verbindungsauflaufs soll dabei für jede Komponente identisch abgewickelt werden, siehe Abbildung (32). Um eine entsprechend stabile Reaktionszeit der teilnehmenden Roboter zu garantieren, sollen zum Start des Verbindungsauflaufs wiederholt Kommandos versendet werden. Dies soll eine erhöhte **Central Processing Unit (CPU)** Laufzeit erreichen, um eine Zeitverzögerung zur Laufzeit der Szenarios zu verhindern.

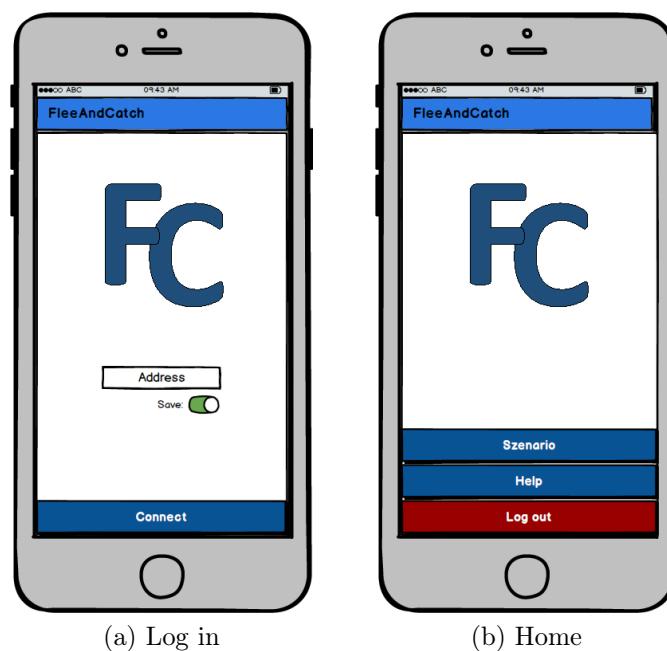


Abbildung 31: Mockup Connection

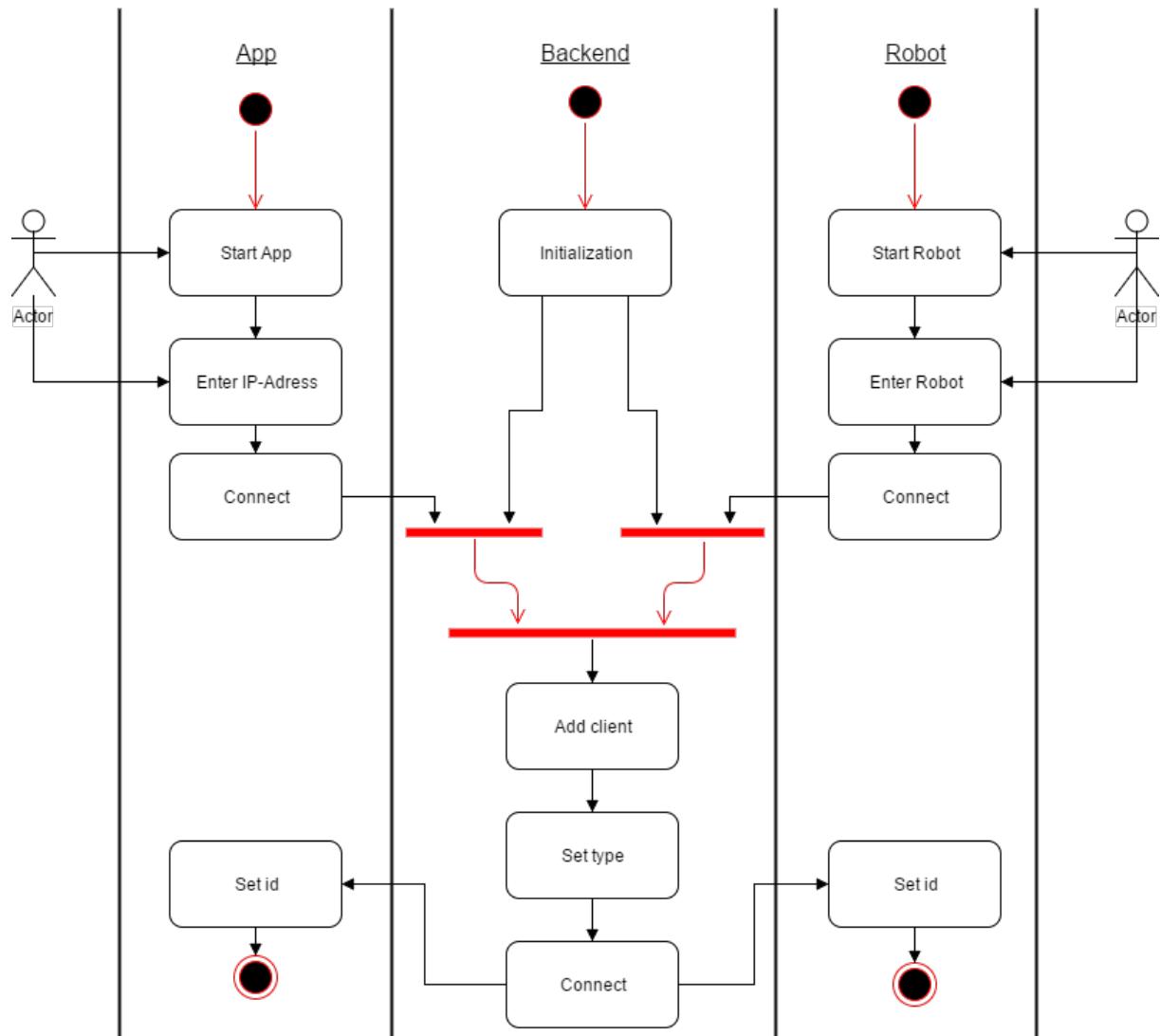


Abbildung 32: Use Case Connection

### 3.5.2 Synchronization

Der Use Case Synchronization stellt den Datenaustausch der beteiligten Komponenten dar, siehe Abbildung (34) und (35). Hierbei sollen verschiedene Typen unterschieden werden, wobei der komplette Datensatz in Form von allen Szenarien, Robotern, oder eines einzelnen Szenarios, sowie Roboters übertragen werden soll. Dies wird einerseits der Erstellung eines Szenarios dienen, als auch dessen Beobachtung durch den Spectator Modus. Die Übertragung eines einzelnen Roboter dient der Aktualisierung der jeweiligen Daten der Desktopanwendung, sowie der App, um diese laufend aktuell zu halten. Die synchronisierten Daten sollen des Weiteren auf den Komponenten über eine UI dargestellt werden können, um die Veränderung der aktuellen Daten zu verdeutlichen und somit eine verbesserte Steuerung zu schaffen.

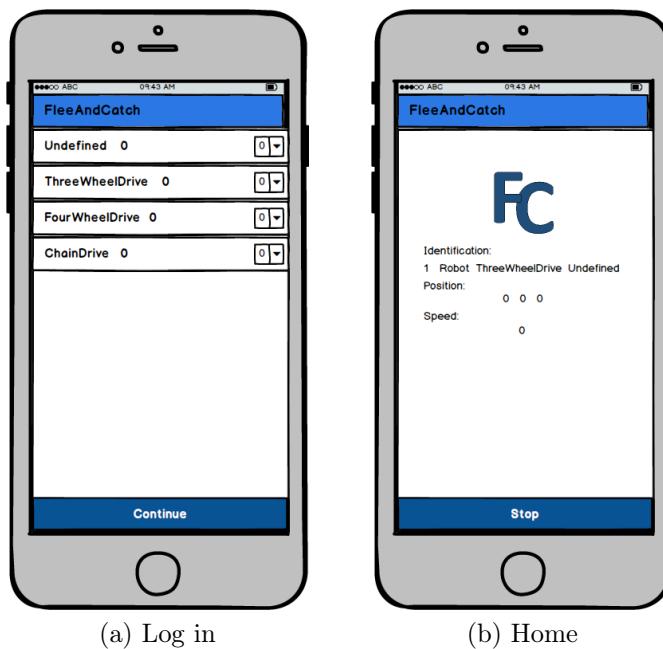


Abbildung 33: Mockup Synchronization

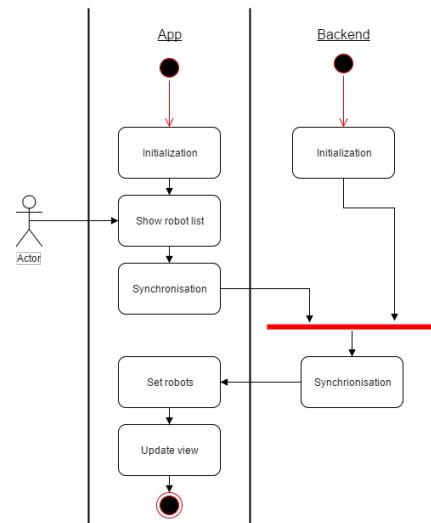


Abbildung 34: Use Case Synchronization

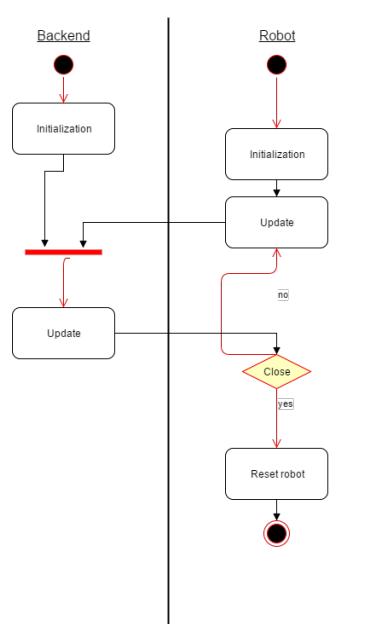


Abbildung 35: Use Case Update

### 3.5.3 Szenario

Der Use Case Szenario stellt den Ablauf der definierten Szenarien, zur Steuerung der Roboter dar, siehe Abbildung (37). Der Nutzer soll dabei zu Beginn das gewünschte Szenario, sowie die teilnehmenden Roboter festlegen. Anschließend wird das Szenario durch das Senden eines Kommandos gestartet, welches von der Desktopanwendung initialisiert wird. Dieses Kommando soll laufend wiederholt werden, um aktuelle Daten, wie Steuerungsinformationen an die Roboter weiterzuleiten. Zur Steuerung sollen hierbei eine direkte von einer positionsorientierten unterschieden werden, wobei die Roboter je nach Kommando die direkten Steuerungsinformationen oder eine anzufahrende Position erhalten.

Diese Implementierung soll den Vorteil einer Auslagerung der Programmlogik schaffen, indem die Ressourcen der Roboter geschont werden und die Logik zentral in der Desktopanwendung ausgeführt werden kann.

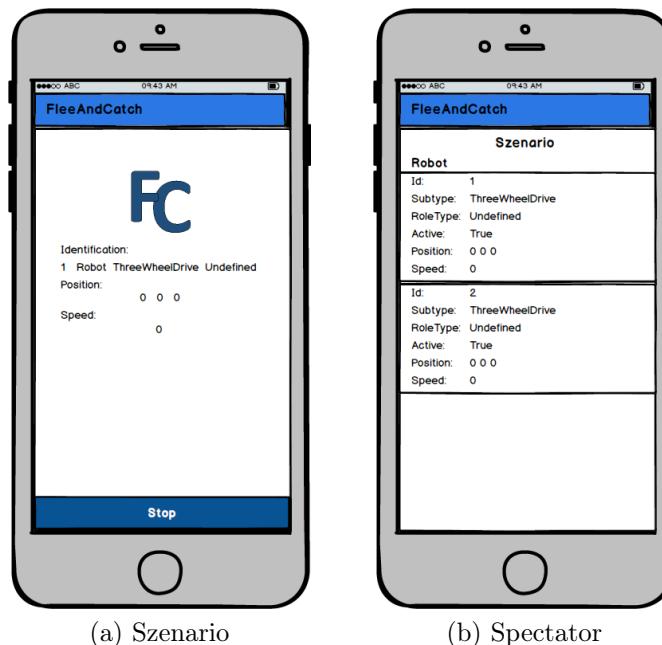


Abbildung 36: Mockup Szenario

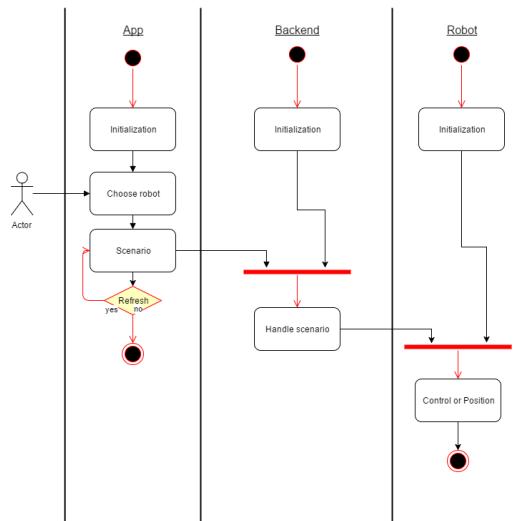


Abbildung 37: Use Case Szenario

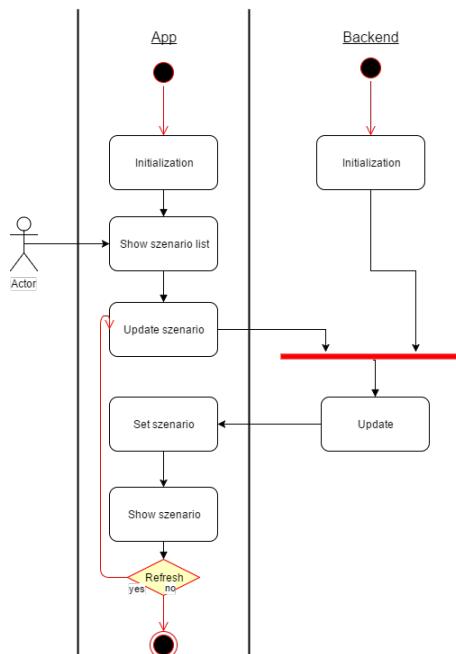


Abbildung 38: Use Case Spectator

### 3.5.4 Exception

Der Use Case Exception stellt den Ablauf eines auftretenden Ausnahmestatus in einem Roboter im Kontext eines Verbindungsverlustes dar, siehe Abbildung (39). Das Backend registriert den Verbindungsverlust zu einem Roboter und sendet eine entsprechende Nachricht an die beteiligten Komponenten des Szenarios. Dies soll ein kontrolliertes Schließen eines Szenarios ermöglichen, welches von einem solchen Ausnahmestatus betroffen ist. Die dabei teilnehmenden Komponenten, wie Nutzer und andere Roboter sollen entsprechend zurückgesetzt werden, damit diese für ein neues Szenario zur Verfügung stehen.

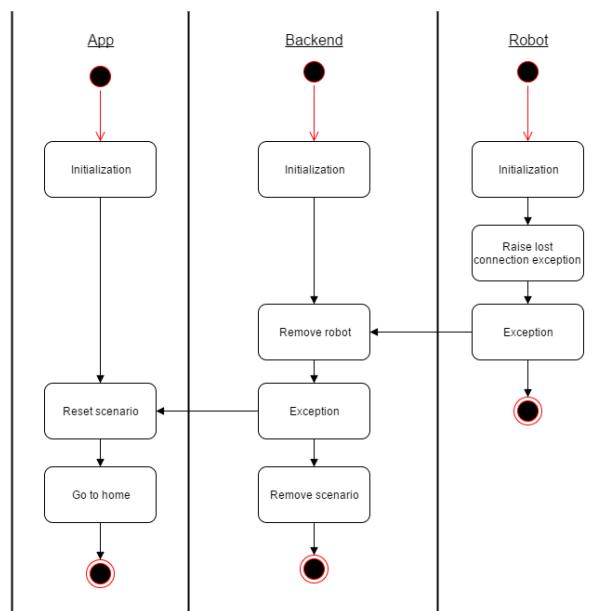


Abbildung 39: Use Case Exception

### 3.6 Kommunikation

Die Kommunikation des Schwarmverhalten basiert auf dem Standard für Netzwerkkommunikation, indem über eine **TCP** Schnittstelle Zeichenketten binär versendet werden. Dabei sollen die zu übertragenen Daten als Zeichenkette serialisiert und binär versendet werden, wobei die Gegenstelle die Daten über eine vorliegende Kommandostruktur des Serialisiert. Um auf die Daten entsprechend zu reagieren, werden diese nach Abhängigkeiten interpretiert. Dies soll über einen Typ realisiert werden, der der jeweiligen Struktur des Kommandos entspricht. Damit die Kommandostruktur als solche von den Komponenten unterschieden werden kann, sollen diese eine zusätzliche Grundstruktur zur Identifizierung der Schnittstelle, als auch den Kommandos erhalten.

**Connection** stellt das Kommando zur allgemeinen Verbindung dar, wobei eine Verbindung sowohl aufgebaut, als auch abgebaut werden kann. Dazu werden Parameter zur Identifizierung von Komponenten genutzt, die entsprechend auch in weiteren Kommandos ihre Anwendung finden. Die Tabelle (4) stellt den allgemeinen Aufbau eines solchen Kommandos dar.

Connection	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete <b>API</b> dar
ClientIdentification identification	Stellt das Identifikationsobjekt der Komponente dar

Tabelle 4: **JSON** Kommando Connection

**Synchronization** stellt das Kommando zur Synchronization der erfassten Daten zwischen den verschiedenen Komponenten dar. Dabei werden die Daten als Listen mit entsprechenden Szenarien und Robotern übertragen. Je nach vorliegendem Typ des Kommandos, können verschiedene Daten übertragen werden, wobei Current einem einzelnen Objekten und die entsprechende Mehrzahl kompletten Datensätzen entspricht.

Synchronization	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete <b>API</b> dar
ClientIdentification identification	Stellt das Identifikationsobjekt der Komponente dar
List scenarios	Stellt die aktuellen Szenarien dar, die übertragen werden
List robots	Stellt die aktuellen Roboter dar, die übertragen werden

Tabelle 5: **JSON** Kommando Synchronization

**Scenario** stellt das aktuell ablaufende Szenario dar, dass als Schwarmverhalten nachgeahmt wird. Darin enthalten sind die teilnehmenden Komponenten, sowie deren aktuell erfassten Daten. Zudem sind Steuerungsinformationen enthalten, die zur Orientierung des Schwarms entsprechend dem Kommando dienen. Die Steuerungsinformationen verändern sich je nach Typ des Szenarios und können in verschiedenen Kontexten kommuniziert werden, um ein unterschiedliches Ergebnis zu erreichen.

Scenario	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete API dar
ClientIdentification identification	Stellt das Identifikationsobjekt der Komponente dar
Scenario scenarios	Stellt das aktuell ablaufende Szenario dar

Tabelle 6: **JSON** Kommando Scenario

**Exception** stellt das Kommando zur Abhandlung eines Fehlverhaltens einer Komponente dar, die anderen Komponenten mitgeteilt werden kann. Sie enthält Informationen zur verursachenden Komponente, sowie dem aufgetretenen Fehler, welcher behandelt wird. Verwendung findet die Fehlerweiterleitung im Verbindungsverlust des Roboters, damit andere Teilnehmer über das Abmelden des entsprechenden Roboters in Kenntnis gesetzt und das Szenario beendet werden kann.

Exception	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete API dar
ClientIdentification identification	Stellt das Identifikationsobjekt der Komponente dar
Exception exception	Stellt die aktuelle Execption dar, die im Roboter auftritt

Tabelle 7: **JSON** Kommando Exception

**Control** stellt das Kommando zur direkten Steuerung eines Roboters dar. Hierbei werden Steuerungsinformationen, wie Geschwindigkeit und Ausrichtung übertragen, um den Roboter entsprechend seiner aktuellen Position relativ zu steuern. Dies findet Anwendung in einer direkten Steuerung, als auch im Schwarmverhalten zur Steuerung des Nutzer abhängigen Roboters.

Control	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete API dar
ClientIdentification identification	Stellt das Identifikationsobjekt der Komponente dar
Robot robot	Stellt den aktuellen Roboter dar
Steering steering	Stellt die aktuellen Steuerungsinformationen dar

Tabelle 8: **JSON** Kommando Control

**Position** stellt das Kommando zur indirekten Steuerung eines Roboters dar. Hierbei werden Positionsdaten übertragen, die der autonomen Navigation des Roboters dienen. Um eine entsprechende Geschwindigkeit zu erhalten, kann zudem ein Geschwindigkeitswert angefügt werden, der in Abhängigkeit der anderen Roboter gesetzt wird.

Position	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete API dar
ClientIdentification identification	Stellt das Identifikationsobjekt der Komponente dar
Robot robot	Stellt den aktuellen Roboter dar
Position position	Stellt die anzufahrende Position dar
Speed speed	Stellt die einzustellende Geschwindigkeit dar

Tabelle 9: **JSON** Kommando Position

## 4 Implementierung

In diesem Kapitel wird die Implementierung des Projektes mit Fokussierung auf die einzelnen Komponenten beschrieben. Da eine komplett detaillierte Beschreibung des ganzen Quellcodes den Umfang dieser Ausarbeitung bei weitem sprengen würde, werden in diesem Kapitel die wichtigsten Implementierungskonzepte deren Funktionsweise, sowie deren Komponenten dargestellt. Für eine vollständig detaillierte Darstellung wird auf den Quellcode und den darin enthaltenen Kommentare sowie das zugehörige github-Wiki (<https://github.com/FleeAndCatch-Dev/FleeAndCatch-Docs/wiki>) verwiesen.

### 4.1 Kommunikation

Die Kommunikation der einzelnen Komponenten des Schwarmverhaltens baut auf einer klar definierten Struktur, um ein Schwarmverhalten als verteiltes System zu ermöglichen, siehe Abbildung (40). Die Daten werden dabei als **JSON** Objekte zur optimalen plattformübergreifenden Interpretation versendet, wobei jeweils die entsprechende Bibliothek zur Serialisierung verwendet wird.

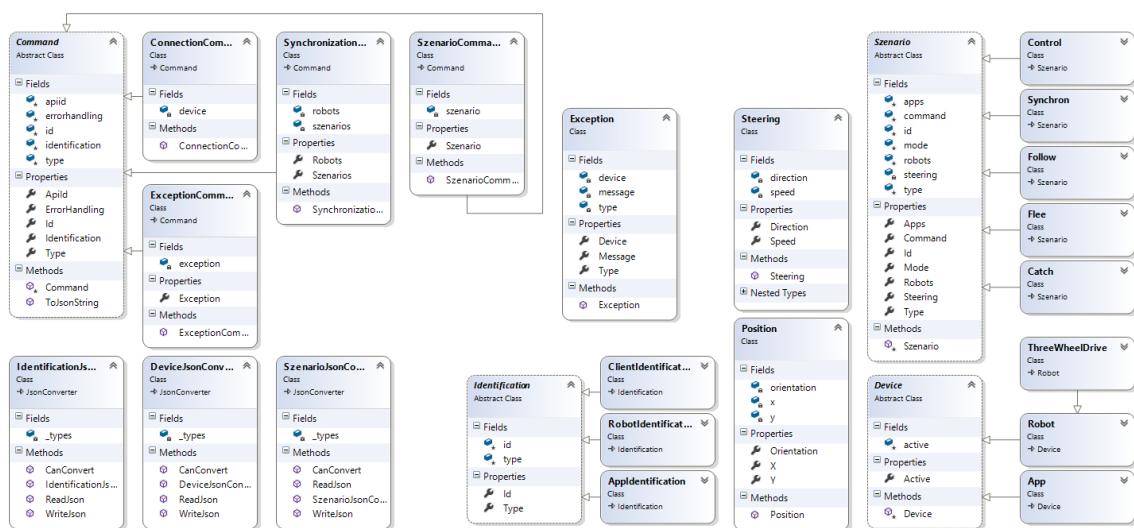


Abbildung 40: Aufbau Commands

Der Kern zur Implementierung der Kommunikation erfolgt in zwei Methoden, die auf jeder Komponente zur Verfügung stehen. Diese dienen zum Versenden, sowie Empfangen von Daten, wobei diese als Zeichenkette serialisiert und in Bytes aufgeteilt werden, siehe Abbildung (41a). Um den vollständigen Umfang der Daten zu erfassen, wird die Größe ermittelt und standardmäßig mittels vier Bytes übertragen. Dadurch ist eine maximale Paketgröße von 32 Byte möglich, was einer Länge von etwa 4 Milliarden Zeichen entspricht. Die Interpretation zum Empfangen erfolgt mit ähnlichem Muster, indem zunächst die Größe der Daten festgestellt wird und die Daten deserialisiert werden, siehe Abbildung (41b).

```
12 references | ThunderSL94, 61 days ago | 1 author, 2 changes
public static async void SendCmd(string pCommand)
{
    if (!connected) throw new System.Exception("There is no connection to the server");
    checkCmd(pCommand);

    var command = Encoding.UTF8.GetBytes(pCommand);
    var size = new byte[4];
    var rest = pCommand.Length;

    //Calculate the length of the data
    for (var i = 0; i < size.Length; i++)
    {
        size[size.Length - (i + 1)] = (byte)(rest / Math.Pow(128, size.Length - (i + 1)));
        rest = (int)(rest % Math.Pow(128, size.Length - (i + 1)));
    }

    try
    {
        //Send the length of the data
        tcpSocketClient.WriteStream.Write(size, 0, size.Length);
        await tcpSocketClient.WriteStream.FlushAsync();

        //Send the full data
        tcpSocketClient.WriteStream.Write(command, 0, command.Length);
        await tcpSocketClient.WriteStream.FlushAsync();
    }
    catch (Exception e)
    {
        throw new Exception(304, "The json command could not send");
    }
}
```

(a) Versende Kommando

```
1 reference | Simon Lang, 18 days ago | 2 authors, 3 changes
private static string ReceiveCmd()
{
    var size = new byte[4];
    byte[] data = null;

    try
    {
        //Read 4 bytes
        tcpSocketClient.ReadStream.Read(size, 0, size.Length);

        //Calculate the full length
        var length = size.Select((t, i) => (int)(t * Math.Pow(128, i))).Sum();
        data = new byte[length];

        //Read the full data
        tcpSocketClient.ReadStream.Read(data, 0, data.Length);
    }
    catch (Exception e)
    {
        throw new Exception(303, "The json command could not receive");
    }

    //Get the string of the data
    var dataString = Encoding.UTF8.GetString(data, 0, data.Length);

    if (!checkCmd(dataString))
        return null;

    return dataString;
}
```

(b) Empfange Kommando

Abbildung 41: Kommunikation

**Kommandos** stellen die Basis der Kommunikationsstruktur sowie den aktuellen Kontext dar, indem sich die Software befindet, siehe Abbildung (42). Sie enthalten grundlegende Attribute zur allgemeinen Identifikation des Kommandos, die zur Interpretation verwendet und über definierte Enums ausgewählt werden. Je nach Kommando sind zusätzliche Objekte enthalten, die durch die jeweilige Id vordefiniert sind.

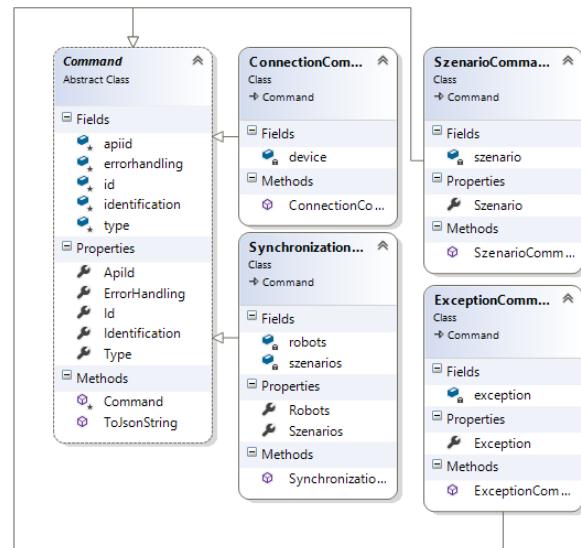


Abbildung 42: Kommandos

**Identifikationen** stellt die individuelle Identität der einzelnen Komponente dar, siehe Abbildung (43). Diese wird durch eine fortlaufende Identifikationsnummer, Typen und je nach Ableitung weiteren Attributen erreicht. Um die jeweiligen Kommandos entsprechend zuzuordnen, sind die Identifikationsobjekte in jedem Kommando vorhanden und bilden die Basisobjekte. Die unterschiedlichen Typen sind dabei für verschiedene Kontexte der Software zuständig. Die ClientIdentification stellt einerseits die Verbindung einer allgemeinen Komponente zur Desktopanwendung dar, wogegen die Robot- bzw. AppIdentification die spezifische Identifikation der Komponente darstellt. Die Erstellung der Identifikation erfolgt jeweils wiederholt zur Anmeldung der Komponente am System. Zunächst wird ein leeres Objekt erzeugt, dass anschließend durch abfragende Kommandos an die entsprechende Komponente befüllt wird, welche hinterher eine berechnete Identifikationsnummer erhält.

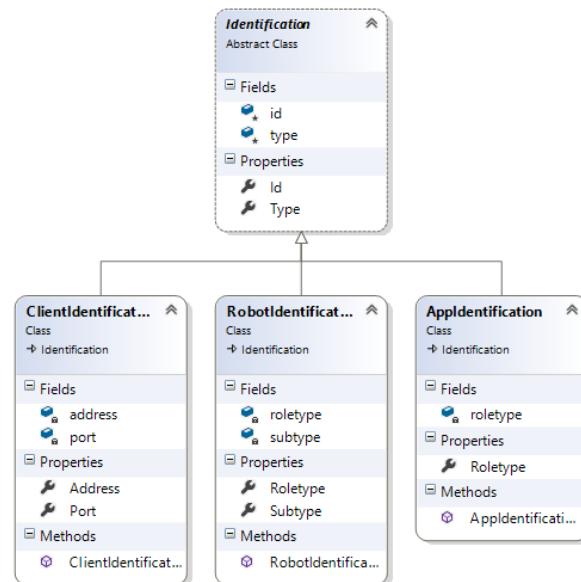


Abbildung 43: Identifikation

Die Erstellung der Identifikation erfolgt jeweils wiederholt zur Anmeldung der Komponente am System. Zunächst wird ein leeres Objekt erzeugt, dass anschließend durch abfragende Kommandos an die entsprechende Komponente befüllt wird, welche hinterher eine berechnete Identifikationsnummer erhält.

**Geräte** stellen die Komponenten dar, die an einem Szenario eines Schwarmverhaltens teilnehmen, siehe Abbildung (44). Sie enthalten jeweils spezifische Identifikations Objekte, zur gegenseitigen Zuordnung, sowie die erfassten Daten der entsprechenden Komponenten. Die Unterscheidung erfolgt in zwei Typen, dem Robot und der App, wobei der Roboter in seine jeweiligen Untertypen gegliedert werden kann.

**Szenarios** stellen den Ablauf des Schwarmverhaltens dar, in dem sich der Nutzer befindet, siehe Abbildung (45). Sie enthalten die jeweiligen Teilnehmer des Szenarios, sowie die Steuerungsinformationen und damit die gesamten Daten des aktuellen Kontextes. Diese Objekte werden laufend aktualisiert und besitzen lediglich zur Laufzeit des Szenarios ihre Gültigkeit. Dabei existieren verschiedene Kategorien von Szenarien, siehe Abschnitt (3.4). Diese definieren jeweils einen unterschiedlichen Kontext und besitzen daher je nach Szenario zusätzliche Attribute.

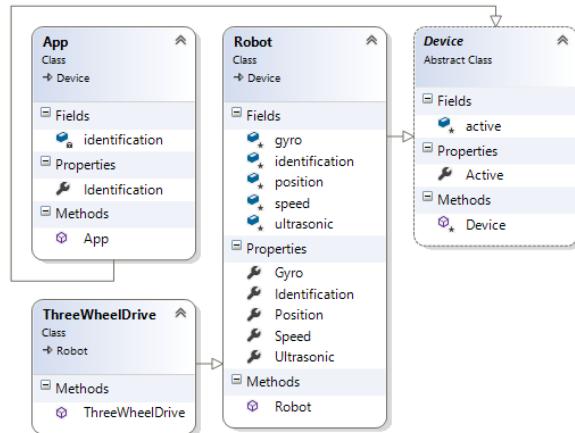


Abbildung 44: Devices

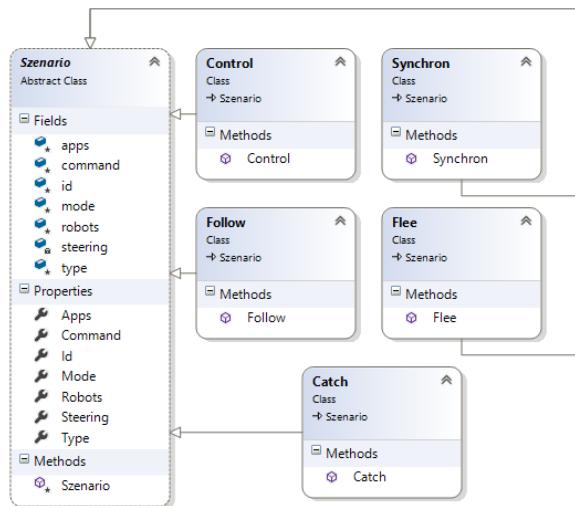


Abbildung 45: Scenarios

**Konverter** dienen der Deserialisierung von abstrakten **JSON** Objekten, welche nicht direkt identifiziert werden können, siehe Abbildung (46). Dazu gehören abstrakte Klassen, sowie Schnittstellen, welche keinem spezifischen Objekt zugeordnet werden können. Die Implementierung erfolgt durch die Überschreibung der entsprechenden Methoden zur Deserialisierung und Serialisierung, siehe Abbildung (47a) und (47b). Je nach Anwendung, wird ein Parameter übergeben, der das Objekt als Zeichenkette beinhaltet. Dieses wird durch eine Abfolge von Bedingungen auf den Typen geprüft, um das Objekt zu erstellen.

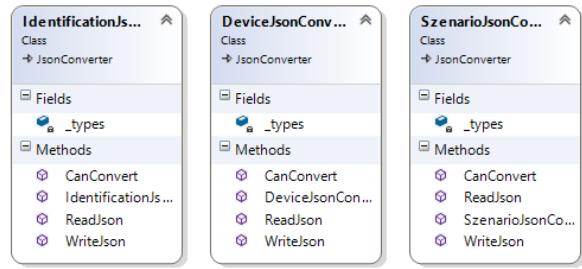


Abbildung 46: JsonConverter

```

2 references | ThunderSL94, 74 days ago | 1 author, 2 changes
public override object ReadJson(JsonReader reader, Type objectType, object existingValue, JsonSerializer serializer)
{
    switch (reader.TokenType)
    {
        case JsonToken.StartArray: //Parse an array
            List<Device> devices = null;
            var jsonArray = JArray.Load(reader);

            foreach (var t in jsonArray)
            {
                if (jsonArray["identification"]["type"] == null) throw new System.Exception("Szenario is not implemented");
                switch (jsonArray["identification"]["type"].ToString()) //Check the identification
                {
                    case "App":
                        devices.Add(t.ToObject<App>());
                        break;
                    case "Robot":
                        devices.Add(t.ToObject<Robot>());
                        break;
                }
            }

            return devices;
        case JsonToken.StartObject: //Parse an object
            Device device = null;
            var jsonObject = JObject.Load(reader);

            if (jsonObject["identification"]["type"] == null) throw new System.Exception("Device is not implemented");
            switch (jsonObject["identification"]["type"].ToString()) //Check the identification
            {
                case "App":
                    device = jsonObject.ToObject<App>();
                    break;
                case "Robot":
                    device = jsonObject.ToObject<Robot>();
                    break;
            }
            return device;
        default:
            throw new System.Exception("Not defined JsonToken");
    }
}

```

(a) ReadJson

```

2 references | Simon Lang, 24 days ago | 2 authors, 3 changes
public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
{
    var t = JToken.FromObject(value);
    if(t.Type != JTokenType.Object)
        t.WriteTo(writer);
    else
    {
        var o = ( JObject)t;
        o.WriteTo(writer);
    }
}

```

(b) WriteJson

Abbildung 47: Device JsonConverter

## 4.2 App

Die Erstellung der **App** erfolgt in einer plattformübergreifenden Implementierung durch Xamarin in C#. Kernelemente stellen hierbei die Struktur, Oberfläche, Businesslogik sowie das Kommunikationssystem dar. Durch die zentrale Verwendung des Kommunikationssystems wird dieses in ein separates Projekt untergliedert, siehe Abbildung (48) und wird als solches von der **App** als Bibliothek eingebunden. Somit lässt sich die Logik einmalig implementieren und kann auf andere Systemen entsprechend übertragen werden.

Die **App** setzt sich aus vier verschiedenen Projekten zusammen, einer **PCL** als Projekt zur plattformübergreifenden Entwicklung und den plattformspezifischen Projekten. Diese werden dem Zielsystem entsprechend ausgewählt und erzeugen durch ihre bestehende Referenz auf die **PCL** einen plattformspezifischen Quellcode als **IL**, der anschließend auf einer Plattform ausgeführt werden kann.

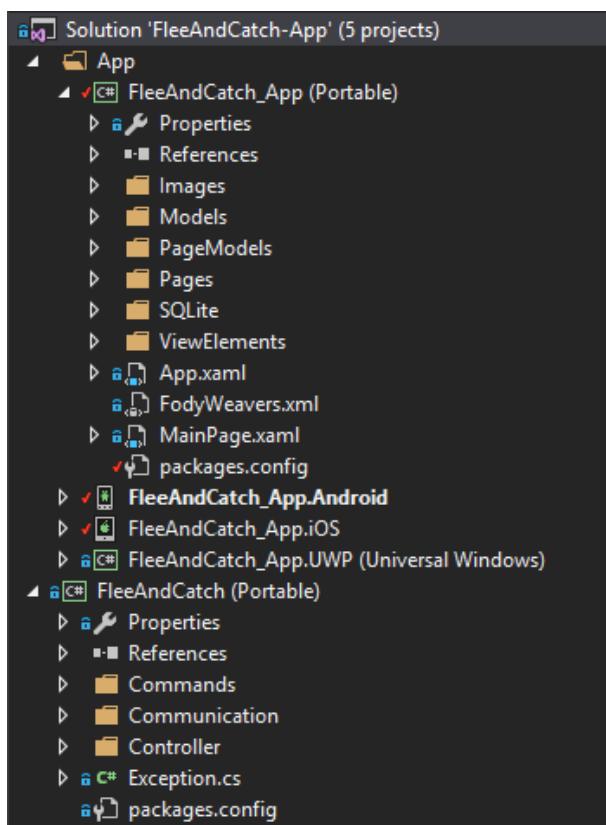


Abbildung 48: Projektstruktur

#### 4.2.1 Workflow

Die Struktur der App basiert auf dem in Xamarin verbreiteten Design Pattern **Model-View-ViewModel**, welches durch ein Benachrichtigungssystem zwischen den verschiedenen Strukturen der App eine Abspaltung der Daten, **GUI** und dem Businesscode erlaubt. Zusätzlich wird die standardmäßig vorhandene CodeBehind Struktur der einzelnen Seiten verwendet, wobei der Businesscode an das Layout der Seite gebunden ist, damit diese miteinander interagieren können. Zur lokalen Speicherung der Daten wird SQLite durch eine implementierte Schnittstelle verwendet, welche plattformübergreifend ansprechbar ist.

**Model-View-ViewModel (MVVM)** stellt ein Design Pattern dar, welches eine grundlegende Struktur im Quellcode ermöglicht, siehe Abbildung (49). Dabei wird die erstellte Benutzeroberfläche von der Logik, sowie den Daten getrennt, um Änderungen unabhängig voneinander durchführen zu können. Die einzelnen Objekte werden dabei durch Referenzen verbunden um diese durch Events entsprechend zu aktualisieren.

- Model:** Datenschicht, welche durch den Benutzer über die **GUI** verändert werden kann und über Datenänderungen die entsprechenden Elemente benachrichtigt
- View:** **GUI** mit den anzuzeigenden Elementen, welche an das ViewModel gebunden sind und der Benutzerinteraktion dienen.
- ViewModel:** Logik des **UI** als zentrale Schnittstelle zwischen dem Model und der View zum Austausch von Informationen, indem entsprechende Methoden und Dienste ausgeführt werden.

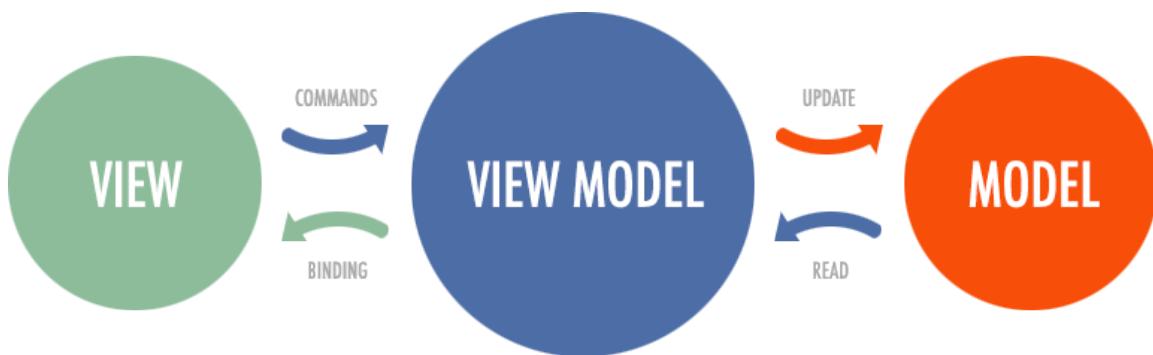


Abbildung 49: **Model-View-ViewModel** [? ]

#### 4.2.2 Graphical User Interface (GUI)

Die Oberfläche der App wird mittels **XAML** durch verschiedene von Xamarin zur Verfügung gestellten Elementen realisiert. Da eine App ein einfaches Bedienkonzept voraussetzt, damit es von jedem beliebigen Benutzer verwendet werden kann, wird in diesem Projekt, auf das Navigationsprinzip der **NavigationPage** gesetzt. Die Navigation wird dabei durch das Aufbauen eines Stapels realisiert, was durch das Prinzip eines Stacks in der Programmierung weit verbreitet ist. Dabei wird durch den Aufruf eines Push auf eine neue Seite gewechselt, wobei der Vorgänger deaktiviert und hinter die neue im Speicher angeordnet wird, siehe Abbildung (50). Dies wird Programm technisch durch eine Liste realisiert, welche langsam aufgebaut wird. Um nun auf den Vorgänger zurück zu gelangen, wird ein Pop verwendet, um die aktuelle Seite sowie deren Daten zu löschen, wobei die vorherige Seite aus dem Speicher geladen wird, siehe Abbildung (51).



Abbildung 50: Wechsel auf neue Seite [? ]



Abbildung 51: Wechsel auf alte Seite [? ]

Für das Layout der einzelnen Seiten wird vor allem auf das **StackLayout** gesetzt. Dieses ist einfach umzusetzen, indem eine vertikale oder horizontale Orientierung zur Anordnung der verschiedenen Elemente festgelegt wird, welche entsprechend ihrer Folge hinzugefügt werden. Zusätzlich können standardmäßige Positionierungen verwendet werden, um das Layout individuell zu beeinflussen.



Abbildung 52: StackLayout [? ]

---

**Die SignIn Page** stellt die Benutzerschnittstelle zur Anmeldung des Nutzers am System dar, siehe Abbildung (53). Dabei gibt dieser die entsprechende IP-Adresse der laufenden Desktopanwendung zur Verbindung an. Durch eine implementierte Logik wird diese IP-Adresse auf ihre Richtigkeit geprüft und die Verbindung gestartet. Bei einer erfolgreichen Verbindung wird der Benutzer zur Hauptseite weitergeleitet, wodurch ihm die entsprechenden Funktionalitäten zur Verfügung stehen. Andernfalls erscheint nach einem definierten Timeout von drei Sekunden eine Fehlermeldung, welche dem User Informationen zum Fehler vorlegt. Durch ein Switch Element ist zusätzlich die Speicherung der IP-Adresse in einer lokalen Datenbank gegeben, wobei diese bei Start der App automatisch geladen und eingefügt wird.



Abbildung 53: SignIn Page

**Die Home Page** stellt die Hauptseite mit der grundlegenden Navigation der App dar, siehe Abbildung (54a). Dabei hat der Benutzer die Möglichkeit ein neues Szenario eines Schwarmverhaltens zu starten, Informationen über die App abzurufen, oder sich vom aktuelles System abzumelden.

**Die Option Page** stellt die Benutzerschnittstelle dar, welche diesen durch die vorhandenen Szenarien anhand eines Carousels leitet, siehe Abbildung (54b). Der Nutzer kann damit ein beliebiges Szenario auswählen und die Konfiguration des Schwarmes beginnen.

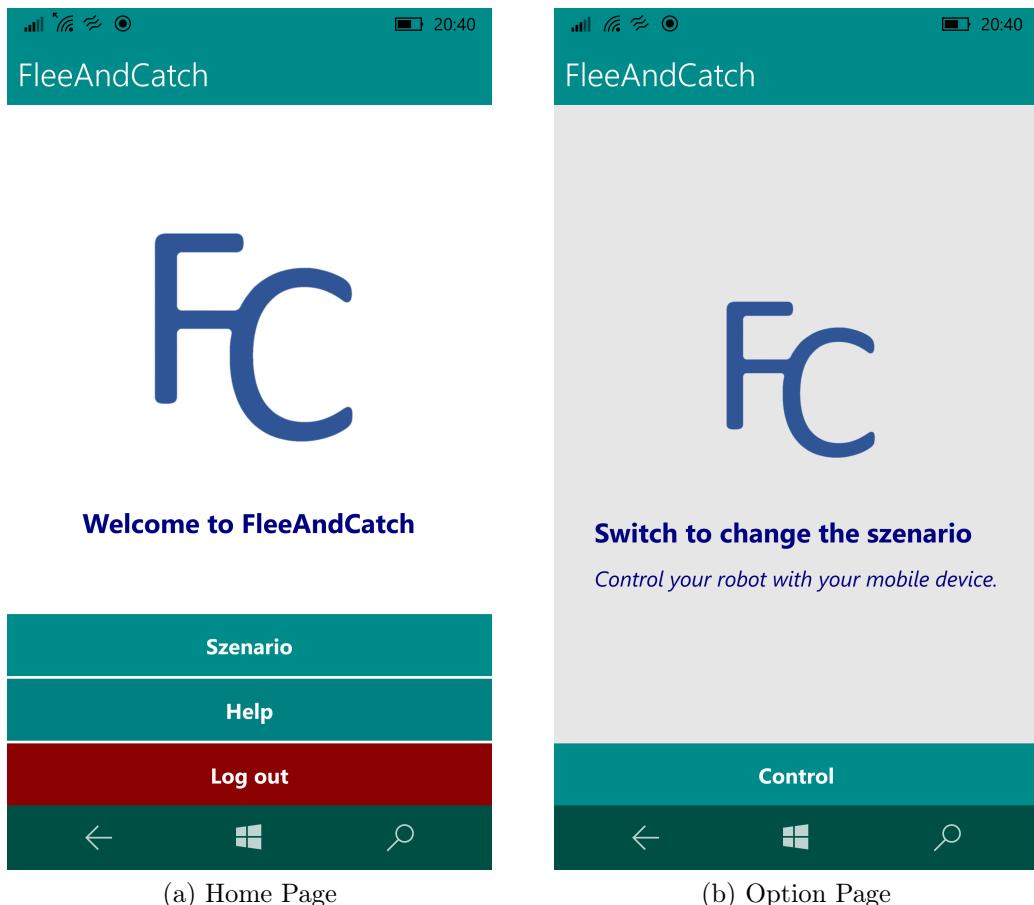


Abbildung 54: Hauptseiten

---

**Die List Page** stellt die Benutzerschnittstelle zur Auswahl der Roboter dar, welche im Szenario involviert sein sollen. Dabei kann der Benutzer unter den verschiedenen Typen wählen, wobei die Anzahl der Roboter vom gewählten Szenario abhängt. Sollte eine ungültige Anzahl an Robotern eingegeben werden, erhält der Benutzer eine Rückmeldung über eine Fehlermeldung und kann dies korrigieren. Bei korrekter Eingabe wird der Benutzer auf eine Seite mit sämtlichen Informationen des Szenarios weitergeleitet und kann dieses starten.

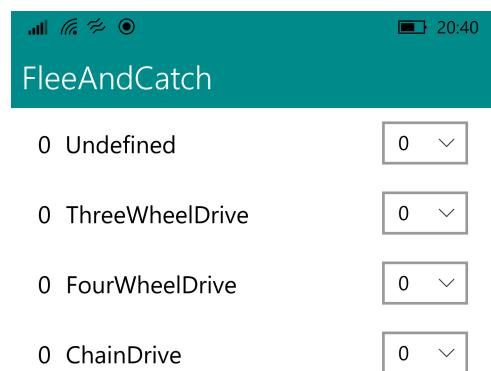


Abbildung 55: List Page

---

**Die Szenario Page** stellt die Benutzerschnittstelle dar, welche einerseits zur Steuerung und der Überwachung des Schwarmverhaltens dient. Dabei werden laufend die Daten der Roboter, die auf dem **GUI** zu sehen sind aktualisiert und entsprechend angezeigt. Die Neigungssensoren des Gerätes, auf dessen Gerät die **App** ausgeführt wird, ermöglichen die Roboter zu steuern, wobei die Richtung und Beschleunigung durch einen Pfeil auf der Oberfläche dargestellt werden. Der Schwarm verhält sich dabei dem entsprechenden Kontext, welchen der Benutzer zuvor eingestellt hat.

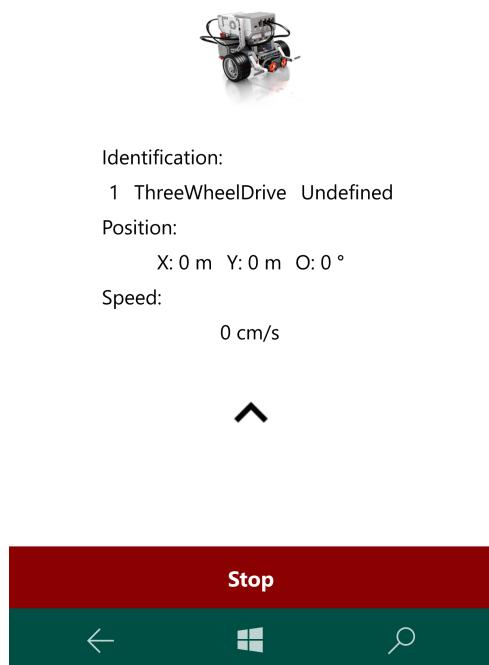


Abbildung 56: Szenario Page

### 4.2.3 Buisnesslogic

Die Logik der App wird mittels des .NET Framework in C# implementiert und ist durch die Strukturierung des MVVM Design Pattern in separate Klassen sowie anhand eines Projektes untergliedert, welches als Bibliothek eingebunden wird. Diese Logik kann in drei Benutzerszenarien gruppiert werden.

1. Anmeldung am System
2. Erstellung eines Szenario
3. Ablauf eines Szenario

**Die Anmeldung am System** beginnt mit der Validierung der vom Nutzer eingegebenen IP-Adresse. Identifiziert wird diese durch die Unterteilung der Zeichenkette in vier Bereiche, welche als Zahl erkannt, und durch einen Punkt abgetrennt sind sowie einen Wertebereich von 0 bis 255 besitzen. Anschließend wird die Verbindung entsprechend konfiguriert, siehe Abbildung (58) und durch den Aufruf der Methode ConnectAsync() gestartet. Dabei wartet die App eine im System fest definierte Zeit von drei Sekunden, siehe Abbildung (57), bis ein Timeout als Fehler geworfen wird, um einen neuen Verbindungsversuch starten zu können.

```
for (var i = 0; i < Default.TimeOut; i++)
{
    if(Client.Connected)
        break;
    await Task.Delay(TimeSpan.FromMilliseconds(10));
}
```

Abbildung 57: Timeout der Verbindung

```
public static void Connect(string pAddress)
{
    ParseAddress(pAddress);

    tcpSocketClient = new TcpSocketClient();
    identification = new ClientIdentification(0, IdentificationType.App.ToString(), pAddress, Default.Port);
    Device = new FleaAndCatch.Commands.Models.Devices.App(
        new AppIdentification(-1, IdentificationType.App.ToString(), RoleType.Undefined.ToString()));
    connected = false;

    if (connected) throw new Exception("Connection to the server is already exist");
    var listenTask = new Task(listen);
    listenTask.Start();
}
```

Abbildung 58: Verbindungsaufbau

Durch die Speicherung der Verbindung in einer lokalen Datenbank mittels SQLite, siehe Abbildung (59) und (60) besteht die Möglichkeit eine einmal eingegebene IP-Adresse beim Start der App wieder zu laden, um diese nicht laufend eingeben zu müssen. Dafür wird eine entsprechende Klasse implementiert, welche die benötigten Daten als Objekt darstellt. Diese Daten werden als Tabellen in der Datenbank ablegt sowie Methoden zu deren Verwaltung bereitgestellt.

```

var connections = SQLiteDB.Connection.GetConnections();
foreach (var t in connections)
{
    t.Save = false;
    SQLiteDB.Connection.UpdateConnection(t);
}

if (Connection.Save)
{
    if (SQLiteDB.Connection.GetConnection(Connection.Address) != null)
    {
        var localConnection = SQLiteDB.Connection.GetConnection(Connection.Address);
        localConnection.Save = true;
        SQLiteDB.Connection.UpdateConnection(localConnection);
    }
    else
        SQLiteDB.Connection.AddConnection(Connection.Address, true);
}

Device.BeginInvokeOnMainThread(() => {
    UserDialogs.Instance.HideLoading();
    var page = FreshMvvm.FreshPageModelResolver.ResolvePageModel<HomePageModel>();
    var navigation = new FreshMvvm.FreshNavigationContainer(page)
    {
        BarBackgroundColor = Color.FromHex("#008B8B"),
        BarTextColor = Color.White
    };
    Application.Current.MainPage = navigation;
});
return;

```

Abbildung 59: Speicherung der IP-Adresse

```

4 references | ThunderSL94, 123 days ago | 1 author, 2 changes
public SQLiteConnection GetConnection()
{
    var sqliteFilename = "fleeandcatch.db3";
    var path = Path.Combine(ApplicationData.Current.LocalFolder.Path, sqliteFilename);
    var connection = new SQLiteConnection(path);
    return connection;
}

```

Abbildung 60: Definition von SQLite

Sobald eine App sich erfolgreich mit dem System verbunden hat, läuft neben dem eigentlichen Hauptprozess, welchen der Nutzer durch seine Eingaben steuert ein Nebenprozess. Dieser wartet auf eingehende Kommandos und interpretiert diese nach einem festgelegten Bedingungen in der Interpreter Klasse, um entsprechend zu reagieren oder den Nutzer zu informieren sowie neue Daten anzugeben.

**Die Erstellung eines Szenario** beginnt mit der Auswahl eines definierten Typ, wobei jeder einen anderen Kontext besitzt, siehe Abschnitt (3.4). Anschließend gelangt der Nutzer zur Auswahl der verschiedenen Roboter, welche im Szenario teilnehmen sollen. Hierbei wird die entsprechende Zahl an Robotern eines Typs einer Liste hinzugefügt, welche der Nutzer durch die GUI auswählt, siehe Abbildung (61). Sollte der Nutzer dabei eine ungültige Anzahl auswählen, erhält er eine entsprechende Rückmeldung, um eine Korrektur vorzunehmen. Bei erfolgreicher Überprüfung wird das Szenario als Objekt erstellt und

---

enthält somit alle Komponenten sowie zusätzliche Informationen zur Identifikation.

```

if (RobotGroupList.Count <= 0) return;
Szenario szenario = null;
var appList = new List<FleeAndCatch.Commands.Models.Devices.App>();
var robotList = new List<Robot>();

//Add the robots
foreach (var t1 in RobotGroupList)
{
    var value = t1.Choosen;
    var type = (RobotType)Enum.Parse(typeof(RobotType), t1.Name);
    while (value > 0)
    {
        foreach (var t in RobotController.Robots)
        {
            if (t.Identification.Subtype != type.ToString() || t.Active) continue;
            robotList.Add(t);
            t.Active = true;
            value--;
            break;
        }
    }
}

```

Abbildung 61: Hinzufügen der Roboter

Um das erzeugte Szenario zu verifizieren, wird das erzeugt Objekt an die Desktopanwendung versendet, siehe Abbildung (62), welche sämtliche beinhalteten Komponenten der Verfügbarkeit überprüft. Bei erfolgreicher Überprüfung wird der Nutzer zum Start des Szenarios mit einem Überblick auf sämtliche Informationen weitergeleitet. Andernfalls wird die Erstellung abgebrochen und der Nutzer kann Änderungen durchführen.

```

//Set the szenario in the client
Client.Szenario = szenario;
SzenarioController.Exist = false;

var cmd = new SzenarioCommand(CommandType.Szenario.ToString(),
    SzenarioCommandType.Init.ToString(), Client.Identification, szenario);
Client.SendCmd(cmd.ToString());

var updateCounter = 0;
while (!SzenarioController.Exist && updateCounter <= 300)
{
    await Task.Delay(TimeSpan.FromMilliseconds(10));
    updateCounter++;
}

```

Abbildung 62: Erstellung des Szenario

**Der Ablauf eines Szenario** teilt sich in verschiedene Prozesse, wobei der Nutzer durch seine Handlungen im **GUI** der Hauptprozess beeinflusst. Daneben existiert ein Prozess zum Datenempfang, welcher Aktualisierungen der Daten des Schwarmes empfängt und entsprechend in der **GUI** einpflegt. Zur Steuerung des Schwarmes, dient ein Prozess, der mittels eines Timer laufend neu gestartet wird und somit immer neue Steuerungskommandos versendet sowie die Oberfläche der **App** aktualisiert.

```

//stop sensors
CrossDeviceMotion.Current.Stop(MotionSensorType.Accelerometer);
//set object active -> false
foreach (var t in Client.Szenario.Robots)
    t.Active = false;
Client.Device.Active = false;
//remove the szenario
SzenarioController.Szenarios.Remove(Client.Szenario);

//Send the control end command
Client.Szenario.Command = ControlType.Undefined.ToString();
var cmd = new SzenarioCommand(CommandType.Szenario.ToString(),
    SzenarioCommandType.End.ToString(), Client.Identification, Client.Szenario);
Client.SendCmd(JsonConvert.SerializeObject(cmd));

Client.Szenario = null;

//navigate to startpage
var page = FreshMvvm.FreshPageModelResolver.ResolvePageModel<HomePageModel>();
var navigation = new FreshMvvm.FreshNavigationContainer(page)
{
    BarBackgroundColor = Color.FromHex("#00888B"),
    BarTextColor = Color.White
};
Application.Current.MainPage = navigation;

return false;

```

Abbildung 63: Abbruch eines Szenario

Sobald ein Szenario abgebrochen wird, müssen sämtliche Prozesse und Daten freigegeben werden, damit der Nutzer anschließend ein neues Szenario starten kann, siehe Abbildung (64). Dafür wird zunächst der Neigungssensor abgeschaltet, da dieser aktuell nicht mehr benötigt wird. Zur Freigabe der Daten werden sämtliche Aktivität der Roboter abgeschaltet sowie das Szenario entfernt. Dem Backend wird anschließend das Beenden des Szenarios mitgeteilt, damit auch dieses die anderen Komponenten informieren sowie seinen reservierten Speicher freigeben kann.

```

switch (Device.OS)
{
    case WinPhone:
    case Windows:
        x = Convert.ToDouble(((MotionVector)e.Value).X.ToString("F"));
        y = Convert.ToDouble(((MotionVector)e.Value).Y.ToString("F"));
        break;
    case Android:
        x = (Convert.ToDouble(((MotionVector)e.Value).X.ToString("F")) / 10) * (-1);
        y = (Convert.ToDouble(((MotionVector)e.Value).Y.ToString("F")) / 10) * (-1);
        break;
    case iOS:
        await CoreMethods.DisplayAlert("Error", "Not supported OS", "OK");
        break;
    case Other:
        await CoreMethods.DisplayAlert("Error", "Not supported OS", "OK");
        break;
    default:
        throw new ArgumentOutOfRangeException();
}

```

Abbildung 64: Neigungssensor

Zur Erzeugung eines neuen Steuerungskommandos, siehe Abbildung (65), werden die Neigungssensoren des mobilen Gerätes verwendet. Da bei der Implementierung der Schnittstelle auf den verschiedenen Plattformen gewisse Unterschiede bestehen, müssen diese in

---

der plattformübergreifenden Programmierung behoben werden. Dies wird durch eine Angleichung der Faktoren sowie ein Vertauschen der Attribute realisiert, wie am Beispiel von Android und Windows. Das Steuerungskommando enthält dabei keine direkten Werte, wie Geschwindigkeit und Winkel, sondern wird durch Zeichenketten (Enums) umgesetzt. Diese verändern ihre Wertigkeit entsprechend der Werten zum Neigungssensor des mobilen Gerätes.

```
Client.Szenario.Command = ControlType.Control.ToString();
Client.Szenario.Steering = new Steering((int)_direction, (int)_speed);

var command = new SzenarioCommand(CommandType.Szenario.ToString(),
    Client.Szenario.Type, Client.Identification, Client.Szenario);
Client.SendCmd(command.ToString());

return true;
```

Abbildung 65: Neues Kommando

## 4.3 Backend

Das Backend ist die Kommunikations- und Verwaltungszentrale des Projekts und bildet das Rückgrat der Kommunikation. Es verwaltet die Devices im Kontext der einzelnen Szenarien und sorgt für den Datenaustausch zwischen den verschiedenen Geräten.

Realisiert ist des Backend wie auch der Roboter in der Programmiersprache Java. Das bringt neben der plattformunabhängigen Lauffähigkeit des Programms noch weitere Vorteile mit sich. So können vor allem Programmkomponenten welche die Kommunikation und den Datenaustausch betreffen für die Roboter- bzw. Backend-Implementierung wiederverwendet werden und müssen nicht komplett neu implementiert werden. Im Folgenden werden die zentralen Komponenten des Backends vorgestellt.

## 4.4 Aufbau

### 4.4.1 Server

Zur Realisierung der Kommunikation verfügt das Backendprogramm über eine Klasse namens Server. Innerhalb dieser Klasse ist ein Server-Socket implementiert welches den Endpunkt einer TCP-Verbindung darstellt Abschnitt siehe X.X. Dieses Server-Socket wird nach dem Programmstart durch aufrufe der Methode `open()` mit IP-Adresse und Port initialisiert und kann anschließend darüber Nachrichten (z.B. Aufbauanfragen für eine TCP-Verbindung) entgegennehmen.

Die Abbildung X zeigt die Funktion `open()` welche eine Server-Socket initialisiert und den listenerThread startet. Nach der Initialisierung des Server-Socket wird ein Thread

```

51  * Method that identifies the IP addresses of the machine []
52  * 
53  * 
54  * 
55  * 
56  * 
57  * 
58  * 
59  * 
60  * 
61  * 
62  * 
63  * 
64  * 
65  * 
66  * 
67  * 
68  * 
69  * 
70  * 
71  * 
72  * 
73  * 
74  * 
75  * Method that initialize a SeverSocket and starts listening for connections */
76  * 
77  * 
78  * 
79  * 
80  * 
81  * 
82  * 
83  * 
84  * 
85  * 
86  * 
87  * 
88  * 
89  * 
90  * 
91  * 
92  * 
93  * 
94  * 
95  * 
96  * 
97  * 
98  * 
99  * 
100 * 
101 * 
102 * 
103 * 
104 * 
105 * 
106 * 
107 * 
108 * 
109 * 
110 * 
111 * 
112 * 

```

Abbildung 66: `open()`-Methode der Sever-Klasse im Backend

gestartet (listenerThread), welcher zyklisch die Methode `listen()` aufruft. Die Aufgabe dieses Threads ist es kontinuierlich auf Verbindungsauftaktwünsche durch Roboter oder Apps zur warten und bei deren eintreffen diese zu verarbeiten.

Die Abbildung X zeigt die Methode `listen()` in der ankommende Verbindungsauftaktwünsche verarbeitet werden. Trifft ein Verbindungsauftaktwunsch ein wird für diesen, vom

```

153*   /* Method that waits for a connection request and process it */
161*   private static void listen() throws IOException {
162
163       opened = true;           //Set flag that server socket is ready!
164
165       while(opened){
166
167           ViewController.setStatus(Status.Waiting);    //Show waiting status on view!
168
169           //Wait if the server socket received a connection request:
170           Socket socket = serverSocket.accept();
171
172           if(socket != null){
173
174               try {
175                   //Made connection optimizations:
176                   socket.setTcpNoDelay(true);
177                   socket.setSendBufferSize(1000);
178                   socket.setReceiveBufferSize(1000);
179               } catch (SocketException e) {
180                   System.out.println("104 " + e.getMessage());      //Error handling!
181               }
182
183               final int id = generateNewClientId();          //Generate a unique client id!
184
185               //Create a new thread that establish the connection and handle it in future:
186               Thread clientThread = new Thread(new Runnable() {
187                   clientThread.start();
188
189               }
190               else
191                   System.out.println("103 " + "The socket is null");
192
193       }
194   }

```

Abbildung 67: `listen()`-Methode der Sever-Klasse im Backend

Server-Socket ein neues Socket Objekt zurück liefert welches ab sofort eine unabhängige Verbindung zum anfangenden Device repräsentiert und ein Senden von Nachrichten in beide Richtungen ermöglicht.

Anschließend werden auf diesem Socket-Objekt noch ein paar Optimierungen durchgeführt um die Latenz der Kommunikation mit dem Device möglichst gering zu halten. Dazu wird zum einen der Nagle-Algorithmus deaktiviert, siehe Abschnitt X sowie entsprechende Puffergrößen festgelegt.

Danach wird ein Thread erzeugt der ab sofort die Kommunikation dieser Verbindung verwaltet. Dazu wird dieser an ein neues Client-Objekt gekoppelt, welches das zugehörige Socket-Objekt aufnimmt. Mit der Erzeugung dieses Client-Objekts der Übergabe des Socket-Objekts an diese und die Zuordnung des entsprechenden Thread zur Verbindungsverarbeitung ist die Verbindung zu einem Device vollständig initialisiert. Ab sofort wird das entsprechende Device (App oder Robot) durch das Client-Objekt repräsentiert und kann durch dieses angesprochen werden. Dazu werden sämtliche Clients in deiner Array-Liste gespeichert und sind durch eine eindeutige ID identifizierbar.

#### 4.4.2 Client

Neben der Server Klasse ist die Client Klasse die zentrale Komponente zum Empfangen und Senden von Daten an die einzelnen Devices. Während die Server-Klasse auf Verbindungsauflaufwünsche der Geräte die sich am Backend anmelden wollen warten, repräsentieren die Clients eine Device inklusive einer vollwertige TCP-Verbindung.

#### 4.4.3 Interpreter

Die Interpreter Klasse ist zuständig für die Interpretation der am Backend eintreffenden Datenpakete der verschiedenen Devices. Dazu enthält jedes Client-Objekt eine eigene Instanz der Klasse Interpreter, so dass eine parallele Verarbeitung ohne Beeinflussungen oder Wartezeiten möglich ist.

Die Interpretation der Datenpakete besteht aus zwei grundlegenden Schritte:

1. Parsen des als String vorliegenden Datenpakets in ein JSON-Objekt
2. Weiterverarbeitung des JSON-Objekts anhand des vorliegenden Kommandotyps

Den ersten Schritt realisiert die Methode `public void parse(String pCommand)`. Zur Umwandlung des vorliegenden Strings verwendet die Methode dazu eine Funktionalität der Bibliothek »org.json«. Anschließend wird das geprägte JSON-Object abhängig von seinem Kommandotyp durch eine entsprechende Methode weiterverarbeitet.

Die Abbildung X zeigt den Quellcode des Parse-Vorgangs in der Methode Vorgangs `public void parse(String pCommand)`.

```

 73  /* Method that parse an incoming command string: */
 74  public void parse(String pCommand) {
 75      JSONObject jsonCommand = null;
 76      CommandType id = null;
 77
 78      try {
 79          //Parse String to JSON-Object with org.json.JSONObject:
 80          jsonCommand = new JSONObject(pCommand);
 81          //Get the type-id of the command:
 82          id = CommandType.valueOf((String) jsonCommand.get("id"));
 83
 84          //If command does not originate from a trustworthy source:
 85          if(!Objects.equals("@fleeandcatch@", (String) jsonCommand.get("apiid"))) {
 86              System.out.println("114 " + "Wrong apiid");
 87          }
 88      }
 89
 90      //If an error during the parse process:
 91      } catch (JSONException e) {
 92          System.out.println("113 " + e.getMessage());           //Error handling!
 93      //If an object does not exist:
 94      } catch (NullPointerException e) {
 95          ViewController.printDebugLine("ERROR: " + e.getMessage() + " / " + e.getStackTrace().toString()); //Error handling
 96      }
 97
 98      //processing of the JSON-Command depending on the command type:
 99  }
```

Abbildung 68: Ausschnitt der `parse()`-Methode der Interpreter-Klasse im Backend

Die folgende Auflistung zeigt die Methoden der Interpreter Klasse:

- `public void parse(String pCommand)` – Diese Methode führt den eigentlichen Parse-Vorgang durch um das als String vorliegende Datenpaket (pCommand) in ein JSON-Objekt zu konvertieren.

- `private void connection(JSONObject pCommand)` – Verarbeitet ein als JSON-Objekt vorliegendes Datenpaket vom Typ Kommandotyp Connection.
- `private void synchronization(JSONObject pCommand)` – Verarbeitet ein als JSON-Objekt vorliegendes Datenpaket vom Typ Kommandotyp Synchronization.
- `private void szenario(JSONObject pCommand)` – Verarbeitet ein als JSON-Objekt vorliegendes Datenpaket vom Typ Kommandotyp Szenario. Dazu existieren folgenden folgende Methoden die die Daten abhängig vom vorliegenden Szenario verarbeiten:
  - `private void szenarioControl(SzenarioCommand pCommand)`
  - `private void szenarioSynchron(SzenarioCommand pCommand)`
  - `private void szenarioFollow(SzenarioCommand pCommand)`
- `private void exception(JSONObject pCommand)` – Verarbeitet ein als JSON-Objekt vorliegendes Datenpaket vom Typ Kommandotyp Exception.

Aus der Auflistung ist gut ersichtlich das es für jeden Kommandotyp eine eigene Methode zu dessen Verarbeitung implementiert wurde. Da diese immer nach dem selben Prinzip die Datenverarbeitung vornehmen wird im folgenden exemplarisch auf eine dieser Methode eingegangen um dieses Prinzip zu verdeutlichen.

#### 4.4.4 Device & Szenario Controller

Neben den für die Kommunikation wichtigen Komponenten spielen bei der Realisierung des Backends auch die Controller-Klassen eine zentrale Rolle, da diese wichtige Verwaltungsfunktionen bereit stellen. Sie sind für die Verwaltung der verschiedenen Devices und Szenarien zuständig.

Dazu existieren die folgenden Controller-Klassen:

- AppController – Der AppController verwaltet alle am Backend angemeldeten Apps.
- RobotController – Der RobotController verwaltet alle am Backend angemeldeten Roboter.
- ScenarioController – Der AppController verwaltet sämtliche aktiven Szenarien.

Die beiden Klassen AppController und RobotController bestehen im wesentlichen aus einer Listen-Datenstruktur (ArrayList), in welcher sämtliche Apps bzw. Roboter die am Backend angemeldet sind gespeichert werden. In dieser Datenstruktur werden die Devices durch Objekte entsprechender Klassen repräsentiert.

---

Die Controller-Klassen verfügen über Datenstrukturoperationen zum Einfügen, Entfernen, Suchen oder zum Aktualisieren von Elementen so das durch entsprechende Methoden eine einfacher Zugriff auf die Devices sowie deren Manipulation möglich ist.

Da alle drei Controller-Klassen mit statischen Datenstrukturen und Methoden arbeiten werden die angemeldeten Devices zentral gespeichert und Verwaltet was einen einfachen und schnellen Zugriff auf diese ermöglicht und Redundanzen und Inkonsistenzen vermeidet. Durch die Verwendete Listen-Datenstruktur ist ein einfaches Suchen, Einfügen sowie Löschen möglich. Zudem kann leicht über die einzelnen Devices iteriert werden was hilfreich ist wenn die selben Daten an mehrere Devices gesendet werden müssen.

Die SzenarioController Klasse verfügt über die gleiche Listen-Datenstruktur zur Speicherung von Szenarien, jedoch bietet sie neben den Einfachen Operatoren noch weitere Komplexere Funktionalitäten zum ...

#### 4.4.5 Die GUI

Neben den rein Funktionalen Komponenten des Backends verfügt dieses auch über eine grafische Benutzeroberfläche die Informationen zu den angemeldeten Geräten und den existierenden Szenarien zur Verfügung stellt. Diese Benutzeroberfläche ist so realisiert das diese komplett losgelöst von den der eigentlichen Funktionalität des Backend ist. Sie setzt lediglich auf dem eigentlichen Programm auf und ist zu dessen Ausführung nicht notwendig. Dadurch ist es möglich das Programm komplett ohne grafische Benutzeroberfläche zu starten so das es im Hintergrund laufen kann und lediglich als Verwaltungs- und Kommunikationsservice dient. Ob die grafische Benutzeroberfläche aktiviert ist oder nicht wird beim Programmstart durch entsprechenden Parametern festgelegt.

Der Nutzen der GUI bestand für uns hauptsächlich im Monitoring und bei der Identifikation sowie dem Aufspüren von Fehlern während der Entwicklung.

Realisiert ist die grafische Benutzeroberfläche mit dem JavaFX-Framework welches Bestandteil der Java-Bibliothek ist uns somit wie Java ebenfalls plattformunabhängig lauffähig ist. Zu ihrer Ansteuerung dient eine spezielle Klasse namens »ViewController« welche lediglich Informationen des eigentlichen Programms entgegen nimmt und an die eigentliche Benutzeroberfläche (View) weiterreicht sofern diese aktiviert wurde. Umgekehrt werden jedoch keine Benutzereingaben von der GUI an das Programm weitergereicht, sondern dienen lediglich zur Manipulation der Anzeige so das dieses komplett unabhängig ist. Lediglich das Beenden des kompletten Programms ist über die grafische Benutzeroberfläche möglich. Abbildung X.X zeigt einen Screenshot der grafischen Benutzeroberfläche des Backends.

Da die grafische Benutzeroberfläche jedoch nicht zur eigentlichen Funktionalität des Backends beiträgt soll an dieser Stelle nicht weiter eingegangen werden.

## 4.5 Roboter

Die Implementierung des Roboters stellt das Kernstück bei der Realisierung dieses Projekts dar.

### 4.5.1 Threads

Für die Umsetzung der Szenarien ist es erforderlich das der Roboter verschiedene Dinge parallel erledigt. Zum einen muss die ständige und zeitnahe Kommunikation mit dem Backend aufrecht erhalten werden, um zum einen Steuerdaten sowie die Positionsdaten der anderen Roboter für die Bewegungsberechnung zu empfangen. Andererseits müssen über diese Verbindung die eigenen Daten übermittelt an das Backend übermittelt werden. Darüber hinaus muss die eigentliche Steuerung des Roboters erfolgen.

Zur Realisierung all dieser Aufgaben sind innerhalb des Roboters die folgenden X Threads implementiert:

- Hauptthread (main thread)
- Kommunikationsthread (connectionThread)
- Steuerungsthread (steeringThread)
- Datenerfassungsthread (synchronizeThread)

In den folgenden Abschnitten werden die einzelnen Threads sowie ihre Aufgaben dargestellt:

**Hauptthread** Der Hauptthread ist der Thread welcher beim Programmstart (Aufruf der Funktion `public static void main(String[] args)`) automatisch erzeugt wird und der immer vorhanden ist. Durch diesen Thread werden in der Initialisierungsphase alle anderen Threads die zur Kommunikation, Steuerung und Datenerfassung benötigt werden erzeugt.

### Kommunikationsthread

**Steuerungsthread** Der Steuerungsthread hat wie der Name schon sagt die Aufgabe den Roboter zu steuern das heißt seine Geschwindigkeit und Bewegungsrichtung umzusetzen. Implementiert ist der Steuerungsthread in der `RobotController` Klasse die als Schnittstelle zwischen der Kommunikation und den Roboterfunktionalitäten fungiert.

Die konkrete Umsetzung der Steuerung die durch diesen Thread erfolgt wird im Abschnitt X.X beschrieben.

---

**Datenerfassungsthread** Neben dem Empfangen von Daten und der Umsetzung der Steuerung muss der Roboter auch seine eigenen Daten wie Position, Orientierung und Geschwindigkeit kontinuierlich erfassen und an das Backend übermitteln so dass diese Informationen den anderen Robotern zur Verfügung gestellt werden können. Da dieser Thread dazu auf Funktionalitäten des Roboters zugreift ist auch dieser innerhalb der Roboter-Controller Klasse implementiert.

Die Realisierung der Datenerfassung ist weit weniger komplex als die der Steuerung des Roboters. Die Abbildung X.X zeigt das Zusammenspiel der einzelnen Threads bzw. den Ablauf ihrer Erzeugung und Terminierung:

#### 4.5.2 Klassen

Neben den verschiedenen Threads enthält das Roboterprogramm zahlreiche Klassen durch welche die vielfältigen Funktionen abgebildet werden die, der Roboter zur Realisierung der verschiedenen Szenarien mitbringen muss.

**Klassenstruktur** Zur Strukturierung und einer sauberen Trennung der einzelnen Klassen und ihren Sichtbarkeiten, sind diese in verschiedenen Paketen (Java-Packages) organisiert. Diese Klassenstruktur ist hierarchisch aufgebaut, orientiert sich an den grundlegenden Bestandteilen des Programms (Kommunikation, Roboterkontrolle etc.) und granuliert in einzelnen Unterpakete anhand der Objekten welche die jeweiligen Klassen darstellen bzw. welche Funktionen diese bieten.

Bei der Umsetzung wurden die Klassen in den folgenden vier großen Paketbereiche strukturiert.

- Kommunikation (`flee_and_catch.robot.communication`) – Enthält sämtliche Klassen die für die Kommunikation mit dem Backend notwendig sind dazu zählen die Eigentlichen Kommunikator (Sockets), die verschiedenen Kommandos sowie serialisierbare Datenobjekte.
- Roboter (`flee_and_catch.robot.robot`) – Hier befinden sich alle Klassen die direkten Zugriff bzw. Einfluss auf den physischen Roboter haben. Dies sind hauptsächlich Sensoren, das Roboter-Interface, Klassen die einen konkreten Roboter darstellen, sowie die zentrale Steuerklasse »RoboterController«.
- View (`flee_and_catch.robot.view`) – Enthält Klassen die der Anzeige d.h. Ansteuerung des LCD-Displays des Roboters dienen. Zwar ist das LCD-Display auch Bestandteil des Roboters, es wurde sich aber für eine Trennung dieser beiden Bereiche entschieden da diese möglichst unabhängig voneinander bleiben sollten.
- Konfiguration (`flee_and_catch.robot.configuration`) – Zentrales Paket welches Klassen zur Konfiguration der verschiedenen Programmteile enthält.

---

Innerhalb dieser Hauptpakete existieren zahlreiche weitere Unterpakete welche die Klassen weiter strukturieren. Die saubere Einordnung und Trennung der Programmklassen ermöglicht es in Verbindung mit der Festlegung entsprechender Sichtbarkeiten, die Klassen gegeneinander abzuschotten. So wird erreicht, dass die Programmteile lediglich über die vorgesehenen Schnittstellen miteinander interagieren und die einzelnen Klassen nur diejenigen sehen und ansprechen können die sie tatsächlich benötigen. Zudem hilft es dem Programmierer bei der Orientierung und dem Programmverständnis.

In den nächsten Abschnitten werden die wichtigsten Klassen und Interfaces des Programms näher beschrieben und ihre Rolle bei der Realisierung des durch die Roboter abgebildeten Schwarmverhaltens erläutern.

**Das Roboter Interface** Das Roboter Interface dient als Implementierungsvorlage für Klassen die einen Roboter in seiner jeweiligen physischen Gestalt d.h. mit der jeweiligen Antriebsform, seinen verbauten Sensoren etc darstellen. Innerhalb dieses Interfaces sind sämtliche Funktionalitäten und Methoden definiert, die ein konkreter Roboter bzw. die entsprechende Klasse, die diesen repräsentiert implementieren muss um durch das Programm gesteuert und in die entsprechenden Szenarien integriert werden zu können.

Innerhalb dieses Interface werden dazu verschiedene Methoden zur Steuerung sowie Abfrage von Roboterparametern definiert. Die durch diese Methoden dargestellten Funktionalitäten sind von so grundlegender Natur das diese durch jeden Roboter realisiert bzw. umgesetzt werden können, wenn auch abhängig von seiner Bauart auf andere Weise.

Folgende Auflistung gibt einen Überblick über die wichtigsten diese Methoden und erläutert kurz die durch sie zur realisierende Funktion.

- `void forward()` – Soll den Roboter geradeaus vorwärts fahren lassen.
- `void backward()` – Soll den Roboter geradeaus rückwärts fahren lassen.
- `void turnLeft()` – Soll den Roboter sich nach rechts oder linkes bewegen lassen abhängig von der übergebenen Richtung (Direction-Objekt).
- `void stop()` – Soll den Roboter anhalten.
- `void increaseSpeed()` – Soll die Geschwindigkeit des Roboters erhöhen (Iterativ bei jedem Aufruf).
- `void decreaseSpeed()` – Soll die Geschwindigkeit des Roboters verringern (Iterativ bei jedem Aufruf).
- `boolean isMoving()` – Soll true zurückgeben wenn sich der Roboter bewegt.
- `Position getPosition()` – Soll die aktuelle Position und Orientierung des Roboters in einem speziellen Position-Objekt zurückgeben.

- 
- `float getRealSpeed()` – Soll tatsächliche (nicht eingestellte) Geschwindigkeit des Roboters zurückgeben.
  - `Robot getJSONRobot()` – Soll alle wichtigen Roboterdaten einem speziellen Roboter-Objekt zurückgeben welches serialisierbar ist und damit via JSON-Objekt an das Backend übertragen werden kann.

Durch die Vorgaben und Verwendung dieser grundlegenden und universellen Funktionalitäten für die Programm-, Szenarien- und Steuerlogik ist es möglich eine generische und flexible Implementierung zu schaffen. Da ein Roboter immer als Objekt dieses Interfaces betrachtet und angesprochen wird ist das restliche Programm vollkommen unabhängig von der konkreten Ausprägung des jeweiligen Roboters und kennt diese nicht mal. Es beschränkt sich bei der Interaktion mit dem Roboter auf die in diesem Interface definierten Methoden.

Der Vorteil dieser Implementierung liegt darin, dass durch diese generische Programmierung lediglich eine neuen Klasse die dieses Interface implementiert notwendig ist um einen neuen Art von Roboter zu Realisierung und in das Projekt mit sämtlichen Szenarien zu integrieren. Dadurch ist das Programm leicht erweiterbar und es steigert seine Wieder verwendbarkeit.

Wie diese konkrete Implementierung der durch diese Methoden definierten Funktionen zu realisieren ist hängt natürlich vom jeweiligen Roboter und seinem Aufbau ab und muss durch den Programmierer entsprechend umgesetzt werden.

**Die ThreeWheelDrive Klasse (Eine konkrete Roboter Klasse)** Die Klasse »ThreeWheelDrive« repräsentiert den „Standard“-Roboter im Projekt. Sie bildet den in Abbildung X.X zu sehenden Roboter mit seinen relevanten Komponenten wie Sensoren, Motoren sowie relevanten geometrischen und physischen Parametern ab. Die Klasse implementiert das Roboter-Interface mit den dort definierten Methoden.

**Die RoboterController Klasse** Die »RobotController«-Klasse übernimmt eine zentrale Rolle bei der Umsetzung der Roboter-Steuerung. Die Klasse stellt die Schnittstelle zum physischen Roboter dar. Sämtliche Steuerkommandos und Sensorabfragen werden über diese Klasse realisiert. Durch diese einheitliche Schnittstelle wird es möglich die Steuerung und ... unabhängig von dem konkreten vorhandenen Roboter (Dreirädriger, Vierrädriger etc) zu realisieren. Dazu nutzt der RoboterController intern eine Instanz des "RoboterInterfaces" was eine Ansteuerung des Roboters

Die folgende Abbildung zeigt die der verschiedenen Komponenten:

#### 4.5.3 Steuerung

Eine der Hauptaufgaben des Roboterprogramms ist die Steuerung des Roboters. Dabei muss das Programm die folgende zwei grundlegenden Steuerungsarten unterscheiden und realisieren.

- Direkte Steuerung – Bei der direkten Steuerung erhält das Programm über das Backend von der App direkte Steuerbefehle wie Rechts, Links, Schneller, Langsamer welche das Programm dann in die entsprechende Bewegung des Roboters umsetzen muss.
- Indirekte Steuerung – Bei der indirekten Steuerung bekommt das Programm keinen konkreten Steuerbefehle sondern muss basierend auf den Positionsdaten der anderen Roboter und dem vorherrschenden Szenario die notwendigen Steuerbefehle des Roboters berechnen.

Dabei erfolgt das Empfangen und Verarbeiten der jeweiligen Daten jeweils zyklisch in einem konfigurierbaren Zeitintervall. Beide Steuerungsarten basieren auf dem grundlegenden Prinzip das die Daten (Steuerbefehle oder Positionsdaten) durch den X-Thread erfasst und gespeichert werden. Der Steuerungsthread wiederum liest diese Daten aus und führt die entsprechenden Roboterbefehle aus.

#### Allgemein

#### Realisierung beim ThreeWheelDrive

#### 4.5.4 Datenerfassung

---

## 5 Evaluation

Durch die Durchführung dieses Projektes zum Thema „Konzeption und Implementierung eines Schwarmverhaltens von mobilen Kleinrobotern anhand eines Verfolgungsszenarios“ entstand eine Software, welche.

Während des Projektes traten verschiedene Komplikationen auf, welche diese auf unterschiedliche Weise beeinflussten. Hier eine Auflistung der bedeutenden Problemfälle:

- Keine Funktionalität von Bluetooth
- Verzögerung der Steuerung
- Unübersichtliche Struktur der Implementierung
- Verhalten des EV3

Da zu Beginn des Projektes zwischen den Robotern und dem Backend eine Bluetooth-Verbindung angedacht war, konnte diese leider nicht umgesetzt werden. Das Problem dabei lag an der komplexen Implementierung von Bluetooth auf den Komponenten sowie einer mangelhaften Dokumentation der Bibliotheken. Daher wurde ein Wechsel auf eine Netzwerkkommunikation mit TCP entschieden, wie sie bereits zwischen App und Backend besteht. Dabei konnte die Implementierung teilweise übernommen werden und musste an wenigen Stellen angepasst werden.

Durch eine Verzögerung der Steuerung, die mit dem Versenden der Kommandos im Roboter auftrat, erzeugte eine Verzögerung der Steuerung, welche dieses erschwert. Dies wurde durch das Einfügen eines Initialisierungskommandos bei der Anmeldung des Systems gelöst. Dabei werden laufend Kommandos zwischen den beiden Parteien hin- und hergesendet, damit die CPU-Laufzeit im System des ROboter entsprechend erhöht wird und die damit verbundene Verzögerung so gut wie möglich verschwindet.

Da eine Implementierung zwangsläufig zu Beginn eines Projektes etwas chaotisch abläuft, kommt es dabei zu einer Unübersichtlichkeit des Quellcode. Um dies zu Verbessern wurde das MVVM Design Pattern in der App integriert, um in den einzelnen Teilen unkomplizierte Abänderungen des Quellcodes vorzunehmen.

Da die EV3 Roboter sehr einfach gestrickte Kleinroboter darstellen und eigentlich als Kinderspielzeug gedacht sind, bringen diese einige technische Probleme mit sich, wenn man diese effizient einsetzen will. Beispielsweise bereitet die Installation des Betriebssystems einige Schwierigkeiten, da dieses mittels einer SD-Karte auf den Roboter kopiert wird, um Anwendungen mittels Hochsprachen auszuführen. Dies schlägt oft fehl und benötigt somit viel Zeit und Geduld. Anderseits zeigt sich dieses Problem in der Steuerung der

Roboter, wobei ein Überlauf der Daten bei bestimmten Methoden entsteht und der Roboter sich mit voller Geschwindigkeit in bestimmte Richtungen bewegt, was er aber nicht machen sollte.

Das Ergebnis dieses Projektes stellt ein sehr respektables Ergebnis dar, wobei auf eine sehr abstrakte Implementierung mittels Kommandos geachtet wurde. Dadurch ist dieses System in vielen Punkten erweiterbar und bietet die Möglichkeiten für diverse neue Implementierungen von Logik. Dabei ist das Ziel vollstens erfüllt, wobei ein Verfolgungsszenario mittels eines Schwarmes von Kleinrobotern durchgeführt werden kann. Und wurde durch das Hinzufügen einer GUI der Desktopanwendung übertroffen, welche die Darstellung der erfassten Daten sämtlicher Komponenten ermöglicht.

## 6 Ausblick

Dieses Projekt stellt ein einfaches Schwarmverhalten von Kleinrobotern dar und ist durch seine abstrakte Implementierung mittels verteilten Systemen auf einer großen Bandbreite erweiterbar. Beispiele wären hierfür:

- Datenerfassung und Auswertung des Schwarmverhaltens
- Installation eines neuronales Netzwerkes
- Einsatz der Szenarien zur Umsetzung von Aufgaben

## Anhang