

# Konzeption und Implementierung eines Sachwarmverhaltens 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

15. 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 Sachwarmverhaltens 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</b>	<b>Einleitung</b>	<b>1</b>
1.1	Ausgangslage . . . . .	2
1.2	Zielsetzung . . . . .	2
<b>2</b>	<b>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 . . . . .	7
2.1.5	Antriebsarten . . . . .	9
2.2	LEGO MINDSTORMS . . . . .	12
2.2.1	Das EV3-System . . . . .	12
2.2.2	Der EV3-Stein (Steuereinheit) . . . . .	13
2.2.3	Motoren . . . . .	14
2.2.4	Sensoren . . . . .	14
2.2.5	Programmierung . . . . .	16
2.3	Application (App) Entwicklung . . . . .	18
2.3.1	Native Apps . . . . .	18
2.3.2	Web Apps . . . . .	18
2.3.3	Hybride Apps . . . . .	19
2.3.4	Plattformübergreifende Apps . . . . .	19
2.3.5	Xamarin . . . . .	20
2.3.6	Mono . . . . .	22
2.3.7	.NET Framework . . . . .	22
2.4	TCP-Kommunikation . . . . .	24
2.4.1	Gundlegendes . . . . .	24
2.4.2	Nagle-Algorithmus . . . . .	25
2.4.3	Kommunikationsablauf . . . . .	25
2.4.4	Socket-Programmierung . . . . .	27
2.5	Schwarmverhalten . . . . .	28
2.5.1	Allgemein . . . . .	28
2.5.2	Vorbilder aus dem Tierreich . . . . .	28
2.5.3	Szenarien . . . . .	29
<b>3</b>	<b>Konzeption</b>	<b>31</b>
3.1	Anforderungsdefinitionen . . . . .	31
3.2	Softwarearchitektur . . . . .	32

---

3.3	Steuerung	34
3.4	Szenarien	34
3.5	Use Cases	36
3.5.1	Connection	36
3.5.2	Synchronization	38
3.5.3	Szenario	40
3.5.4	Exception	42
3.6	Kommunikation	43
<b>4</b>	<b>Implementierung</b>	<b>46</b>
4.1	Kommunikation	46
4.2	App	51
4.2.1	Workflow	51
4.2.2	Graphical User Interface (GUI)	51
4.2.3	Buisnesslogic	52
4.2.4	Deployment	52
4.3	Backend	52
4.4	Robot	52
<b>5</b>	<b>Evaluation</b>	<b>53</b>
<b>6</b>	<b>Ausblick</b>	<b>54</b>







# Abbildungsverzeichnis

1	Vergleich Offene und Geschlossene Steuerung . . . . .	8
2	Einachsige Lenkung . . . . .	10
3	Einachsige Lenkung . . . . .	10
4	Dreirad-Antrieb . . . . .	11
5	Einachsige Lenkung . . . . .	11
6	Einachsige Lenkung . . . . .	11
7	Zentrale Komponenten des EV3-Systems . . . . .	12
8	App-Entwicklung [? ] . . . . .	18
9	Xamarin [? ] . . . . .	20
10	Shared Project [? ] . . . . .	20
11	Portable Class Library [? ] . . . . .	21
12	Unterstützung Portable Class Library [? ] . . . . .	21
13	Mono [? ] . . . . .	22
14	.NET Framework Ausführung [? ] . . . . .	23
15	Common Language Runtime [? ] . . . . .	23
16	TCP Verbindungsaufbau . . . . .	26
17	TCP Datenaustausch . . . . .	26
18	TCP Verbindungsabbau . . . . .	26
19	Fischschwarm . . . . .	28
20	Termitenschwarm . . . . .	29
21	Softwarearchitektur . . . . .	33
22	Steuerung . . . . .	34
23	Logo . . . . .	35
24	Mockup Connection . . . . .	36
25	Use Case Connection . . . . .	37
26	Mockup Synchronization . . . . .	38
27	Use Case Synchronization . . . . .	39
28	Use Case Update . . . . .	39
29	Mockup Szenario . . . . .	40
30	Use Case Szenario . . . . .	41
31	Use Case Spectator . . . . .	41
32	Use Case Exception . . . . .	42
33	Aufbau Commands . . . . .	46
34	Kommunikation . . . . .	47
35	Kommandos . . . . .	48
36	Identifikation . . . . .	48
37	Devices . . . . .	49

38	Scenarios . . . . .	49
39	JsonConverter . . . . .	50
40	Device JsonConverter . . . . .	50
41	ModelViewModelView [? ] . . . . .	51

---

## Tabellenverzeichnis

1	Einordnung von Sensoren . . . . .	6
2	Eigenschaften der EV3-Motortypen . . . . .	14
3	Eigenschaften der EV3-Motortypen . . . . .	17
4	JavaScript Object Notation (JSON) Kommando Connection . . . . .	43
5	JSON Kommando Synchronization . . . . .	43
6	JSON Kommando Scenario . . . . .	44
7	JSON Kommando Exception . . . . .	44
8	JSON Kommando Control . . . . .	45
9	JSON Kommando Position . . . . .	45

# 1 Einleitung

Heutzutage werden viele Arbeitsschritte im Industrie-, Dienstleistungs- und Agrarsektor von Maschinen verrichtet, da diese eine effizientere Arbeit leisten und weniger Kosten als Menschen verursachen. Diese Technik erfährt mit der Zeit eine immer stärkere Verbesserung durch einen verbesserten Arbeitsablauf und technische Innovationen. Um diesem laufenden Wandel gerecht zu werden setzen immer mehr Unternehmen auf Industrie 4.0, was aktuell zu einem regelrechten Schlagwort im technischen Bereich wird.

Die zum Einsatz kommenden Maschinen arbeiten dabei meist mit einfachster Technik für die Umsetzung

Um dabei einen optimalen Arbeitsablauf zu schaffen wird auf die Theorie des Schwarmverhaltens zurückgegriffen, indem die zusammen agierenden Maschinen sich gegenseitig im Arbeitsablauf unterstützen und die gegenseitigen Schwächen ausgleichen. Diese Verhaltensstrukturen stammen meist aus der Tierwelt, wie am Beispiel von Fischschwärmen, Ameisen oder Bienen, wobei jedes Individuum des Schwarms seine Aufgaben für ein gemeinsames Überleben des Schwarmes erfüllt.

## 1.1 Ausgangslage

Die Ausgangslage des Projektes stellen verschiedene Maschinen und Roboter ohne intelligentes System dar, welche einen Mehrwert durch eine Dienstleistung oder die Bearbeitung eines Produktes erwirtschaften. In diesem Projekt wird dies dargestellt durch den Einsatz von LEGO Mindstorm EV3 Roboter, welche mittels Struktur orientierter Programmierung verwendet werden können.

- LEGO Mindstorm EV3
- LEGO Mindstorm Sensorik
- Kabellose Netzwerkschnittstelle

## 1.2 Zielsetzung

Das Ziel dieser Studienarbeit stellt die Implementierung eines komplexen Schwarmverhalten mit einem Verfolgungsszenario zwischen Kleinrobotern. Dabei soll ein KOMMunikationssystem aufgebaut werden, auf dessen Grundlage die Komponenten funktionieren. Dabei sollen entsprechende Grundaktionen, wie Bewegen der ROboter und Sensorikansteuerung realisiert werden.

- Kommunikationssystem
- Grundlegende Steuerungsfunktionen
- Abbildung von Schwarmverhalten
- Nutzereinsicht in erfasste Daten

## 2 Theoretische und Technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen des Projektes, die zur Umsetzung nötig sind beschrieben.

Platzhalter kommt noch Text

Platzhalter kommt noch Text

Platzhalter kommt noch Text

Platzhalter kommt noch Text

### 2.1 Robotik

Die Robotik beschäftigt sich mit dem Entwurf, der Konstruktion sowie der Programmierung, Steuerung und dem Betrieb von Robotern

[<http://wirtschaftslexikon.gabler.de/Definition/robotik.html>]. 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 [Mobile Roboter - 2].

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

Die VDI-Richtlinie 2860 von 1990 definiert einen Roboter wie folgt:

*»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 Definition 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, weil die Arbeitsprozess und die Umgebung auf den Roboter zugeschnitten und vollständig bekannt sind [MR 2].

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

### 2.1.2 Mobile Roboter

Mobile Roboter bewegen sich selbständig durch eine sich meist ständig ändernde Umwelt. All ihre Aktionen sind somit von ihrer aktuellen Umgebung abhängig, die in ihrer konkreten 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 selbständig erfassen, diese Sensordaten auswerten und auf Grundlage dessen ihre nächsten Aktionen planen [MR 2].

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 war 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 KI-Forschung im Bereich der Handlungsplanung und dem selbständigen Lernen [MR 5f].
- **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 (mehr als das 30fache) [MR 8f]
- **Stanley:** Stanley ist ein vollständig autonomer Roboter der 2005 am Grand Challenge Wettbewerb teilnahm und diesem 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 Umgebungswahrnehmung 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 durch felsige oder sandige Bereiche sowie durch Wasserläufe führte in knapp unter 7 Stunden.

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 hinsichtlich zwei Aspekten klassifizieren, zum einen über was die Sensoren Informationen liefern (die Umwelt oder den Roboter selbst) und andererseits hinsichtlich ihrer Arbeitsweise. [MR 24]

- **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 Natur aus vorhandenen Signale wie z.B. das einfallende Licht durch eine Kamera.

Die folgenden Tabelle zeigt beispielhaft die Einordnung einiger 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 [MR 26 f.]:

- **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.



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

Tabelle 1: Einordnung von Sensoren

- **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 ermittelten Messwerte gewisse Fehler auf [MR27].

Damit die Robotersteuerung darauf Rücksicht nehmen kann müssen die entsprechenden Kenngrößen der Sensoren bekannt sein. Folgende Größen sind mit Bezug auf Sensorfehler von Bedeutung [MR 27 f.]:

- **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.
- **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 System zu ermöglichen verschiedene Kenngrößen zu ermitteln. Im Folgenden werden einige 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 (siehe Abschnitt X.X) die Geschwindigkeit und Orientierung eines mobilen Roboters zu bestimmen. Zur Drehwinkelmessung dienen Impuls- sowie Inkrementalgeber welche mechanisch, photo-elektrisch und elektro-magnetisch arbeiten [MR 28 ff.].

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 [MR 31]. Dazu werden heute meist piezo-elektronische Gyroskope verwendet welche Dreh- und Linearbeschleunigungen bezüglich aller drei Raumachsen erfassen [MR 32].

- **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 Kompass sowie Inklinometer welche den Neigungswinkel zur Erdanziehrichtung bestimmen, verwendet [MR 32 f.].

- **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 Laserentfernungsmesser. Es gibt aber auch Sensoren die mittels Triangulation (Infrarotsensor) oder dem Verhältnis zwischen ausgesandter und zurückkehrender Energie (Radar) die Entfernung zu Objekten bestimmen [MR 36 f.].

Neben diesen elementaren Kenngrößen gibt es noch zahlreiche weitere die für ein entsprechendes mobile System von Bedeutung sind. Dies ist beispielweise eine globale Positionsbestimmung mittels GPS oder die Detektion 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 jedoch ihre Grenzen nämlich da, wo sich ein Ro-

boter in einer unstrukturierten Umgebung agiert und von Ereignissen oder Parametern abhängt, die nicht kontrollierbar bzw. vorab nicht bekannt sind [MR 3]. Mobile Roboter verfügen über eine geschlossene Steuerung bei der eine Rückkopplung der Umgebung durch Sensorendaten erfolgt. In einem iterativen Verfahren werden die von den Sensoren ermittelten Umgebungsdaten bei der Planung von Steuerbefehlen einbezogen, die resultierenden Aktionen ausgeführt was ggf. die Umgebung verändert, woraufhin der Regelkreis erneut beginnt.

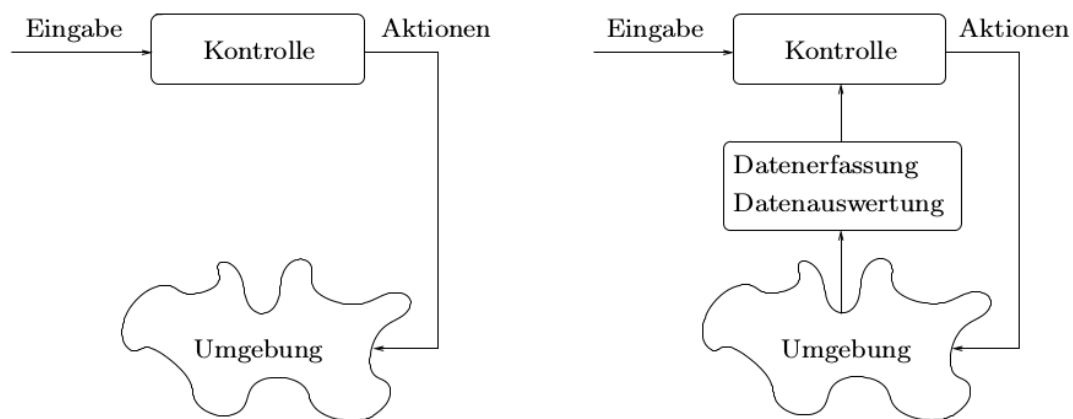


Abbildung 1: Links: Offene Steuerung. Rechts: Geschlossene Regelung mit Rückkopplung

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 [MR 67]

**Odometrie & Koppelnavigation** Die Odometrie bezeichnet eine Methode zur Berechnung von Position und Orientierung eines mobilen Systems anhand von Bewegungsdaten. Räderbasierte Systeme benutzen dafür die Anzahl der Radumdrehungen, während laufende Systeme die Anzahl ihrer Schritte verwenden. Ist beispielsweise der Raddurchmesser 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 [MREPE 98].

Ist der Ausgangspunkt eines Roboters, die zurückgelegte Strecke bzw. Geschwindigkeit

sowie die Fahrtrichtung genau bekannt sind, kann die Navigationskomponente einfach die Bewegung des Robotersystems über die Zeit integrieren und so die aktuelle Position und Orientierung des Roboters bestimmen [HR 107 f.]

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 Interaktionsschritt (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 Interaktionsschritt 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 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 [HR 108].

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 [MREPE 115].

### 2.1.5 Antriebsarten

als Koordinaten (x,y,z-Punkte) Die entscheidende Eigenschaft des mobilen Roboters ist die Fähigkeit sich (selbständig) 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 tau-

chenden Robotern [MR 103].

Um den Umfang dieser Ausarbeitung überschaubar zu halten, beschränken wir uns auf radgetriebene Roboter.

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 [MR 107 f.]

**Differentialantrieb** Dieser Antrieb besteht aus zwei voneinander unabhängig angetriebene, starren Rädern die sich auf einer Achse befinden. Zusätzlich gibt es meist ein oder auch mehrere passiv mitlaufenden Stützrädern 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 diese Antriebs sind seine einfache Mechanik und gute Manövrierbarkeit jedoch erfordert dieser Antrieb eine Radregelung in Echtzeit.

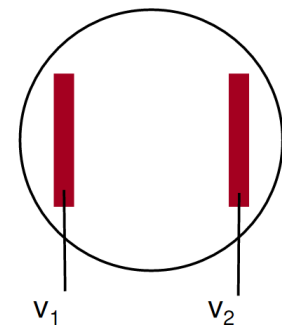


Abbildung 2: Einachsige Lenkung

**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 die beiden Räder einer Achse lenkbar. Für den Antrieb ist es möglich ein der beiden Achsen als auch beide für den anzutreiben.

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

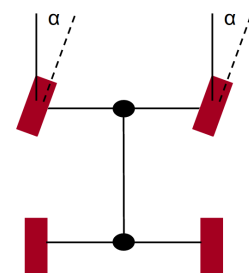


Abbildung 3: Einachsige Lenkung

**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 aber auch möglich, dass ein Motor die beiden Hinterräder antreibt, und das Vorderrad allein der Lenkung dient.

Die einfache Mechanik dieser Antriebsform ist ein Vorteil jedoch ist die Manövrierbarkeit 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.

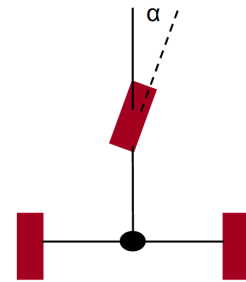


Abbildung 4: Dreirad-Antrieb

**Synchro-Andrieb** Der Synchro-Antrieb hat die selbe Radanordnung wie der Dreirad-Antrieb, jedoch werden die drei Räder synchron angetriebene und sind auch nur synchron drehbare 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 [MR 107 f.].

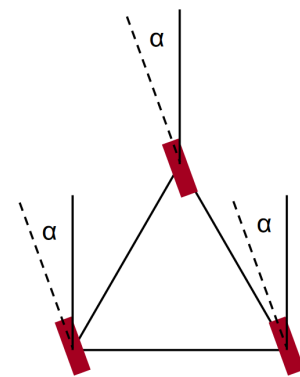


Abbildung 5: Einachsige Lenkung

**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. Im folgenden soll hier der Differenti-

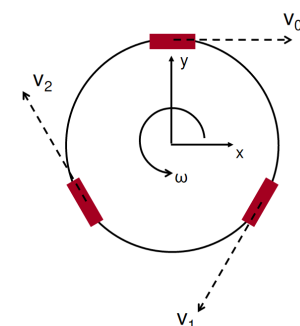


Abbildung 6: Einachsige Lenkung

alantrieb detaillierte behandelt werden da dieser für dieses Projekt von besonderer Bedeutung ist.

## 2.2 LEGO MINDSTORMS

LEGO MINDSTORMS ist eine seit 1988 existierende Produktserie des Spielwarenherstellers LEGO [vgl. ? , 21]. LEGO MINDSTORMS ermöglicht das Bauen, Programmieren und Steuern verschiedener LEGO Roboter. Dies 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 3 Stufe der LEGO MINDSTORMS-Serie zusammen [vgl. ? , Seite 21].

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 [vgl. ? , Seite 22].

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

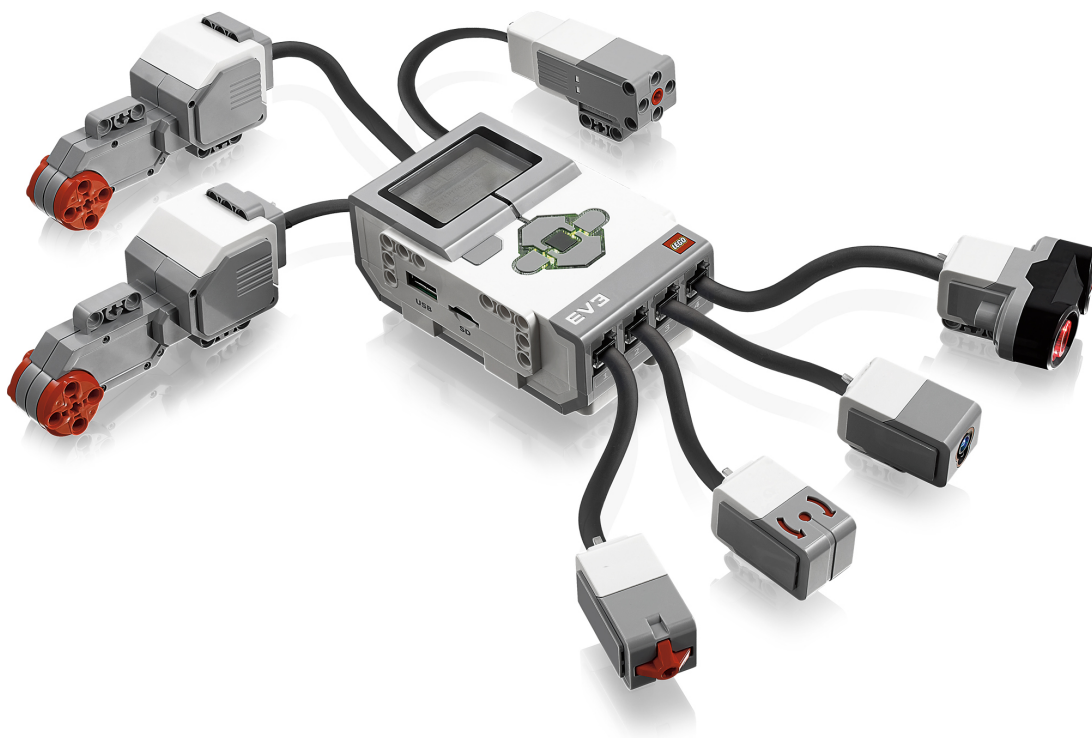


Abbildung 7: Zentrale Komponenten des EV3-Systems



Neben den elektronischen Komponenten gehören auch nicht elektronische Teile wie Verbindungsstücke, Balken und Zahnräder wie sie aus gängigen LEGO Produkten bekannt 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. dieses 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 eine Computer welcher selbständig 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 [vgl. ? , 21].

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 [vgl. ? , Seite 21].

Für den Anschluss von Motoren und Sensoren verfügt der EV3-Stein über 8 Ports, an welche die anderen System-Komponenten müber Kabel mit RJ12-Steckern angeschlossen werden. 4 der Ports dienen für den Anschluss von Motoren, die restlichen 4 Ports für die Abfrage von Sensorwerte [vgl. ? , 21].

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 Programmen angesprochen und ausgewertet werden Start Mein Kanal Trends Abos BIBLIOTHEK [vgl. ? , 21].

Die folgende Auflistung zeigt einige Leistungsmerkmale des EV3-Steins [vgl. ?? , Seite 23 f., Seite 32].

- 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



- 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 [vgl. ? , 92]. Die Motoren lassen sich sehr exakt steuern und ermöglichen so einen synchronen Betrieb mehrerer Motoren [vgl. ? , Seite 29 f.].

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 [vgl. ? , 101]:

- Farbmodus (passiv) - In diesem Modus erkennt der Sensor 7 verschiedenen Farben.
- RGB-Modus (aktiv) - In diesem Modus sendet der Sensor nacheinander rotes, grünes und blaues 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.
- 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 zu folgen.

**Ultraschallsensor** Diese aktive Sensor verwendet 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 [vgl. ? , 32 f.]:

- 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 [vgl. ? , 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 [vgl. ? , 33].

Eigenschaften:

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

**Rotationssensor (Integriert)** Wie bereits im Abschnitt X.X dargelegt verfügen die beiden Motortypen über integrierte Rotationssensoren die es ermöglichen, die Umdrehungen der Motoren auszulesen. Durch diese Sensoren ist es möglich durch Odometrie Rückschlüsse über die Bewegung bzw. Position des Roboters zu schließen.

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 haus eigene 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>1</sup>

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

**leJOS** Das LEGO Java Operating System abgekürzt leJOS ist ein Framework, das 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 [vgl. ? , 21 f.].

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 [vgl. ? , 23 f.].

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 API was unter anderem die Integration von weiteren Sensoren etc. erleichtert [vgl. ? , 23 f.]. Im folgenden sind einige Features die leJOS bietet aufgelistet:

- Objektorientierte Programmierung mit Java

---

<sup>1</sup>[vgl. ? , 25 f.]

Eigenschaft / Program- miersprache	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

- 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-Komponeten
- High-Level-Robotik-Tasks (Navigation, Localization etc.)

## 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, 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 Teile, dem User Interface (UI), das meist mit einer Extensible Markup Language (XML) 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>2</sup>

Vertreter diese 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 aktuellen den höchsten Marktanteil und eine entsprechende Popularität unter Entwickler und Nutzer.

### 2.3.2 Web Apps

Die Entwicklung von web **Apps** arbeitet mit systemübergreifenden Ressourcen und greift auf gängige Webtechnologien, wie Hypertext Markup Language (HTML), Cascading Style Sheets (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 veröffentlicht und aktualisiert werden

<sup>2</sup>[vgl. ? , Unterschiede und Vergleich native Apps vs. Web Apps]

kann, jedoch wird eine stabile Internetverbindung vorausgesetzt.<sup>2</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 Hardwarechnittstellen, um diese mit Application Programming Interfaces (APIs) anzusprechen.<sup>3</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 Betriebssystems 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>3</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 Betriebssysteme, wie Android, iOS, Windows und Windows Phone ausgeführt werden kann. Um nativen Quellcode auf den verschiedenen Systemen auszuführen, setzt Xamarin auf verschiedene Softwarekomponenten, um aus einem mit .NET entwickelten Projekt nativen Quellcode zu erzeugen.



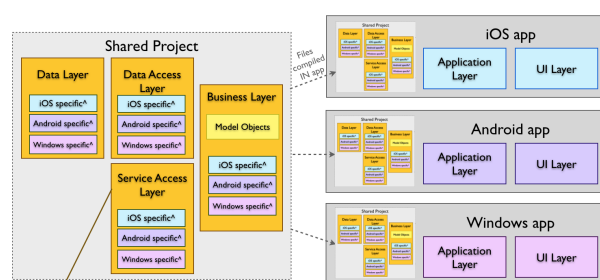
Abbildung 9: Xamarin [?] ]

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>4</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>4</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.

**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 ko-



<sup>4</sup>[vgl. ? , Introduction to Mobile Development - Xamarin]

<sup>4</sup> Hf PLATFORM compiler directives

piert den Quellcode entsprechend in das zu bauende Projekt, siehe Abbildung (10).<sup>5</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.

**Portable Class Libraries** ermöglichen dem Entwickler die Implementierung von plattformübergreifenden Bibliotheken, aus denen **Dynamic Linked Libraries (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 (??).<sup>6</sup> Eine Portable Class Library besitzt darüber hinaus verschiedene Vor- bzw. Nachteile, die für oder gegen ihre Nutzung sprechen.

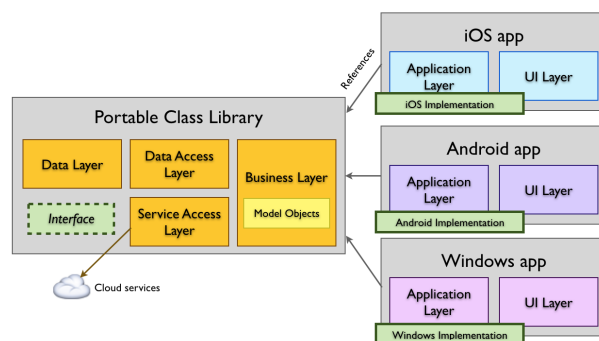


Abbildung 11: Portable Class Library [? ]

Vorteile:

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

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

Nachteile:

- Keine Referenzierung von plattformspezifischen Quellcode
- Keine Standardbibliotheken vorhanden

Abbildung 12: Unterstützung Portable Class Library [? ]

<sup>5</sup>[vgl. ? , Shared Projects - Xamarin]

<sup>6</sup>[vgl. ? , Introduction to Portable Class Libraries - Xamarin]



### 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>7</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 vor-



Abbildung 13: Mono [?] ]

derster Stelle ein Compiler, um den erstellten Quellcode in die jeweilige Maschinsprache 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, die 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.

### 2.3.7 .NET Framework

Das .NET Framework dient zur Entwicklung sowie Ausführung von Anwendungen, die mit Programmiersprachen implementiert werden und auf den Standards von .NET basieren. Es besteht aus verschiedenen Komponenten, wobei der Kern des Frameworks in der CLR liegt.<sup>8</sup> Diese ist verantwortlich für die Laufzeitumgebung und somit für die Ausführung der Anwendungen, indem es die bereitgestellten Ressourcen des Systems nutzt.

---

<sup>7</sup>[vgl. ?, About Mono]

<sup>8</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>8</sup> Anschließend wird er durch eine JIT Kompilierung in einen IL-Quellcode konvertiert, um diesen nativ auf dem System ausführen zu können.<sup>8</sup> Der IL-Quellcode setzt dabei auf die gesetzten Standards der CLI auf, die eine sprach- und plattformunabhängige Entwicklung von Anwendungen ermöglicht.<sup>8</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.

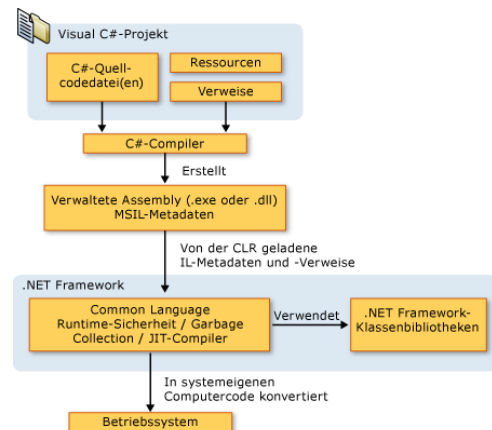


Abbildung 14: .NET Framework Ausführung [? ]

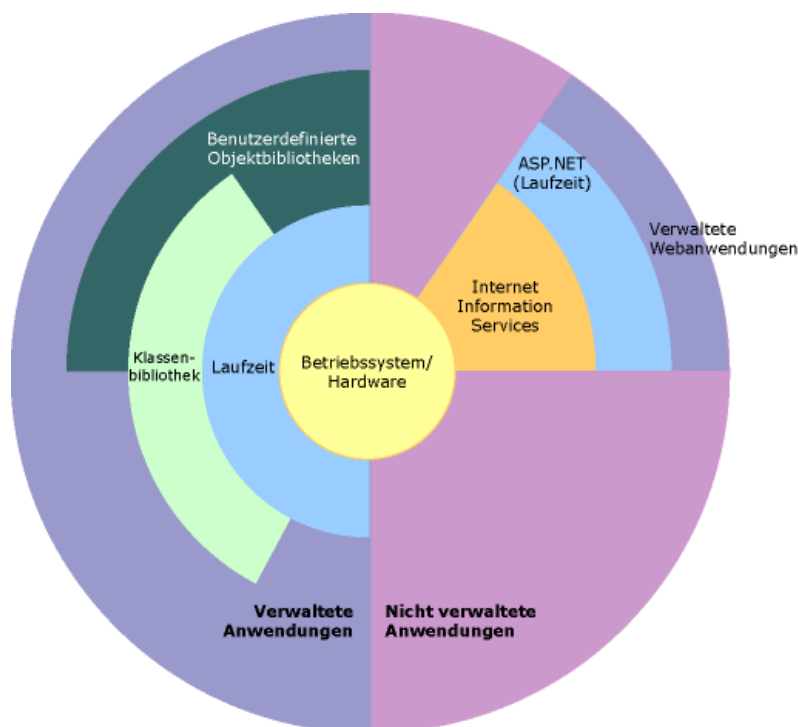


Abbildung 15: Common Language Runtime [? ]

## 2.4 TCP-Kommunikation

Das Transmission Control Protocol (TCP) ist ein Transportprotokoll und ermöglicht einen Datenaustausch zwischen kommunizierenden Anwendungsinstanzen in einer Ende-zu-Ende-Beziehung zwischen. TCP ist als Transportprotokoll auf in der 4. Schicht des OSI-Modells angesiedelt und basiert auf dem Internetprotokoll (IP) mit dem zusammen es 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 UDP das Transportprotokoll, auf dem die meisten Anwendungen im Internet basieren [GD189].

### 2.4.1 Grundlegendes

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 [GD].

TCP nutzt prinzipiell folgende Protokollmechanismen:

- Drei-Wege-Handshake-Verbindungsauf- und -abbau
- Positives, kumulatives Bestätigungsverfahren mit Timerüberwachung für jede Nachricht
- Implizites negatives Bestätigungsverfahren (NAK-Mechanismus): Bei drei ankommenden 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
- 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 versuchte Nagle-Algorithmus aus Optimierungsgründen zu verhindern, dass viele kleine Nachrichten gesendet werden, da dies schlecht für die Netzauslastung ist [GD198].

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 weiteren Daten so lange im Sendepuffer gesammelt bis alle zuvor gesendeten Datenpakete vom Empfänger bestätigt wurden oder so viele Daten im Sendepuffer liegen das 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 allerdings nicht für alle Anwendungsszenarien optimal da es die Latenz erhöht. Insbesondere bei Anwendungen die eine unmittelbare Antwort der Gegenstelle benötigen wie SSH- oder Telnet-Anwendung sorgt dies für Verzögerungen. In diesem Fall ist es besser den Nagle-Algorithmus auszuschalten. [GD198 f.]

### 2.4.3 Kommunikationsablauf

Da es sich bei TCP um ein verbindungsorientiertes Protokoll handelt gliedert sich die Kommunikation in die 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. Nach dem Verbindungsaufbau haben die beiden Rollen jedoch keine Bedeutung mehr und die beide Teilnehmern sind sowohl bei der Datenübertragung als auch beim Verbindungsabbau gleichberechtigt.

#### Verbindungsaufbau

Der Verbindungsaufbau bei TCP basiert auf dem Three-Way-Handshake. Dabei schickt

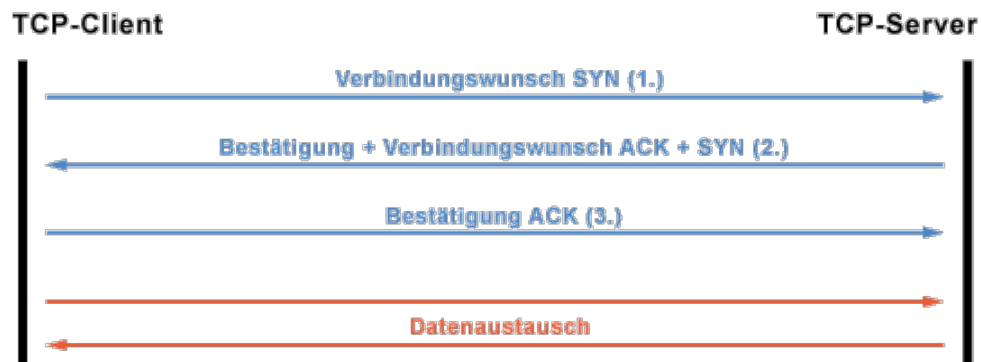


Abbildung 16: TCP Verbindungsaufbau

der 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 Bestätigugsvorgangs ist die Verbindung initiiert und der Datenaustausch zwischen den Teilnehmern kann beginnen [EK].

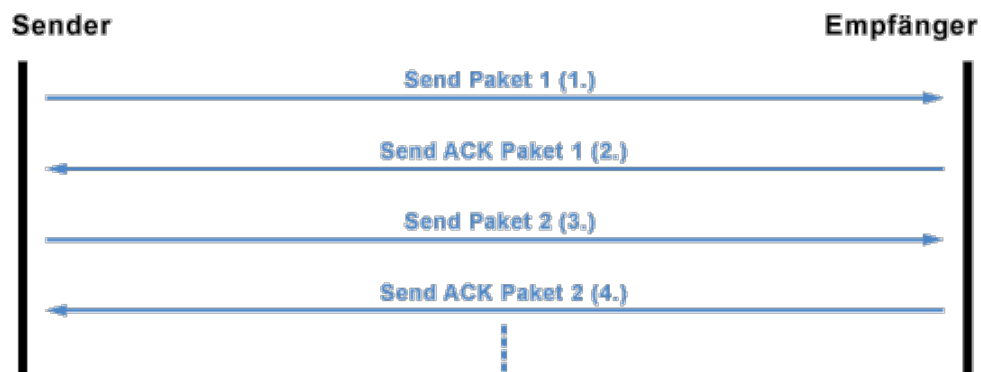


Abbildung 17: TCP Datenaustausch

## Datenaustausch

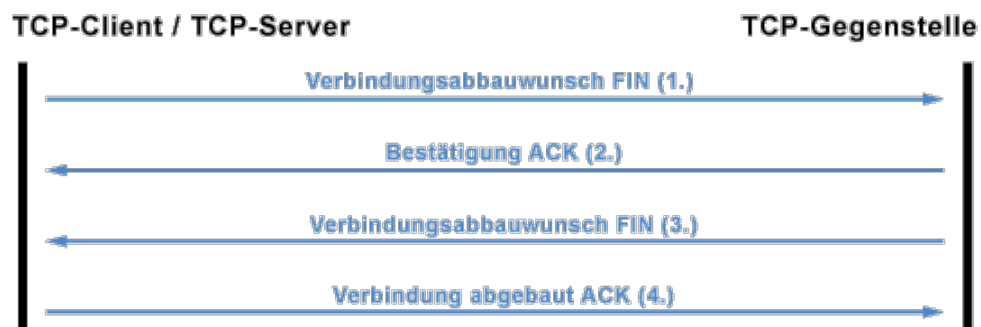


Abbildung 18: TCP Verbindungsabbau

**Verbindungsabbau** Nach Abschluss der Datenübertragung wird von einer Seite (egal von welcher) ein Verbindungsabbau initiiert. Dazu dient einen etwas modifizierten Drei-

Wege-Handshake-Mechanismus. Jede der beiden Verbindungsrichtungen der Vollduplex-Verbindung wird abgebaut, d.h. beide Seiten bauen ihre „Senderichtung“ ab. Die initiiierende 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).

#### **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 [GD].

## 2.5 Schwarmverhalten

Dieser Abschnitt beschäftigt sich mit dem Schwarmverhalten, welches mit Roboter imitiert werden soll.

### 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 im Schwarm, indem diese nicht als Beutetier identifiziert werden können. Andernfalls kann sich der Schwarm als solches effizienter gegenüber Fressfeinden behaupten und Jungtiere schützen. Zudem dient es einer Erleichterung der Fortpflanzung, da in einem Schwarm eine entsprechende Auswahl an Partnern besteht, um eine genetische Vielfalt zu erreichen.

### 2.5.2 Vorbilder aus dem Tierreich

Zur Erstellung eines Schwarmverhaltens existieren verschiedene Ideale, 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 19: 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 Prozent richtet, durch die der Schwarm gesteuert werden kann. 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.

**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 direkten Umgebung, wie dem Hinterlassen von Spuren als Merkmale. Dieses Prinzip ermöglicht auch den kleinsten und primitivsten Organismen einen evolutionären Vorteil zu erlangen, indem sie sich nicht allein den Gefahren stellen, sondern in einer großen Masse zusammenarbeiten.



Abbildung 20: Termitenschwarm

Dies ermöglicht das Bauen riesiger Nester, indem die Insekten ihren inneren Plan, mit ihrer aktuellen Umgebung vergleichen und dadurch intuitiv erkennen, welche Arbeit sie zu erledigen haben. 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 zu errichten. Zudem lässt sich dadurch eine indirekte Kommunikation und Zusammenarbeit zwischen Verschiedenartigen Robotern realisieren.

### 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 Abläufe verbessern.

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 werden und über ein existierendes Netzwerk kommuniziert werden. Bei einer indirekte



Kommunikation analysiert der Roboter dagegen seine direkte Umgebung, um daraus Veränderungen zu registrieren und entsprechend darauf 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
  1. Separation
  2. Aligment
  3. Cohesion
  4. Ausseneinflüsse

## 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. Um die Daten entsprechend zu interpretieren benötigt jede Komponente den jeweiligen Aufbau der Kommunikation, damit diese verwertet und Aktionen ausgeführt werden können.

Diese Aktionen repräsentieren den Grundbestandteil des Schwarmverhaltens und sind auf die 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 des Schwarmverhaltens besteht aus drei Hauptkomponente, den Robotern, einer Desktopanwendung, sowie einer mobilen App, siehe Abbildung 21.

Diese Komponenten kommunizieren über ein drahtloses Netzwerk mittels **Transmission Control Protocol (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 die die kommunizierten Daten serialisiert und als Objekte dargestellt werden können, siehe 3.6.

Die Roboter basieren auf dem Java System **leJos**, da durch die bereitgestellten Bibliotheken für **EV3** Systeme eine unkomplizierte Implementierung von Logik möglich ist, sowie eine direkte Unterstützung von **Eclipse** gegeben ist, um erstellte Software zu debuggen. 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 veranlassen und steuert somit den Schwarm.

Das Backend dient als 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 der einzelnen Komponenten und stellt diese anhand einer auswählbaren Hierarchie dar.



Abbildung 21: Softwarearchitektur

### 3.3 Steuerung

Die Steuerung des Roboterschwarms greift in sämtlich 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 22 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 22: Steuerung

### 3.4 Szenarien

Zum Ablauf der Software greift der Roboterschwarm auf verschieden 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 implementiert 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 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 ihrer Position in der Schlange entsprechend der Position ihres Vordermannes, zu dem diese vollkommen autonom fahren. 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 erwischt 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 23: 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 Verbindungsaufbau von einem Verbindungsabbau unterschieden werden kann. Der Ablauf eines Verbindungsaufbaus soll dabei für jede Komponente identisch abgewickelt werden, siehe Abbildung 25. Um eine entsprechend stabile Reaktionszeit der teilnehmenden Roboter zu garantieren, sollen zum Start des Verbindungsaufbaus wiederholt Kommandos versendet werden. Dies soll eine erhöhte Central Processing Unit (CPU) Laufzeit erreichen, um eine Zeitverzögerung zur Laufzeit der Szenarios zu verhindern.

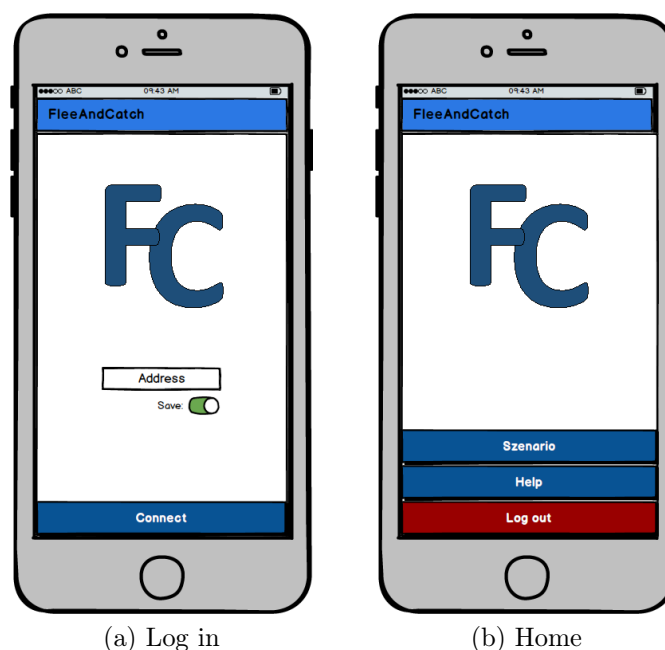


Abbildung 24: Mockup Connection

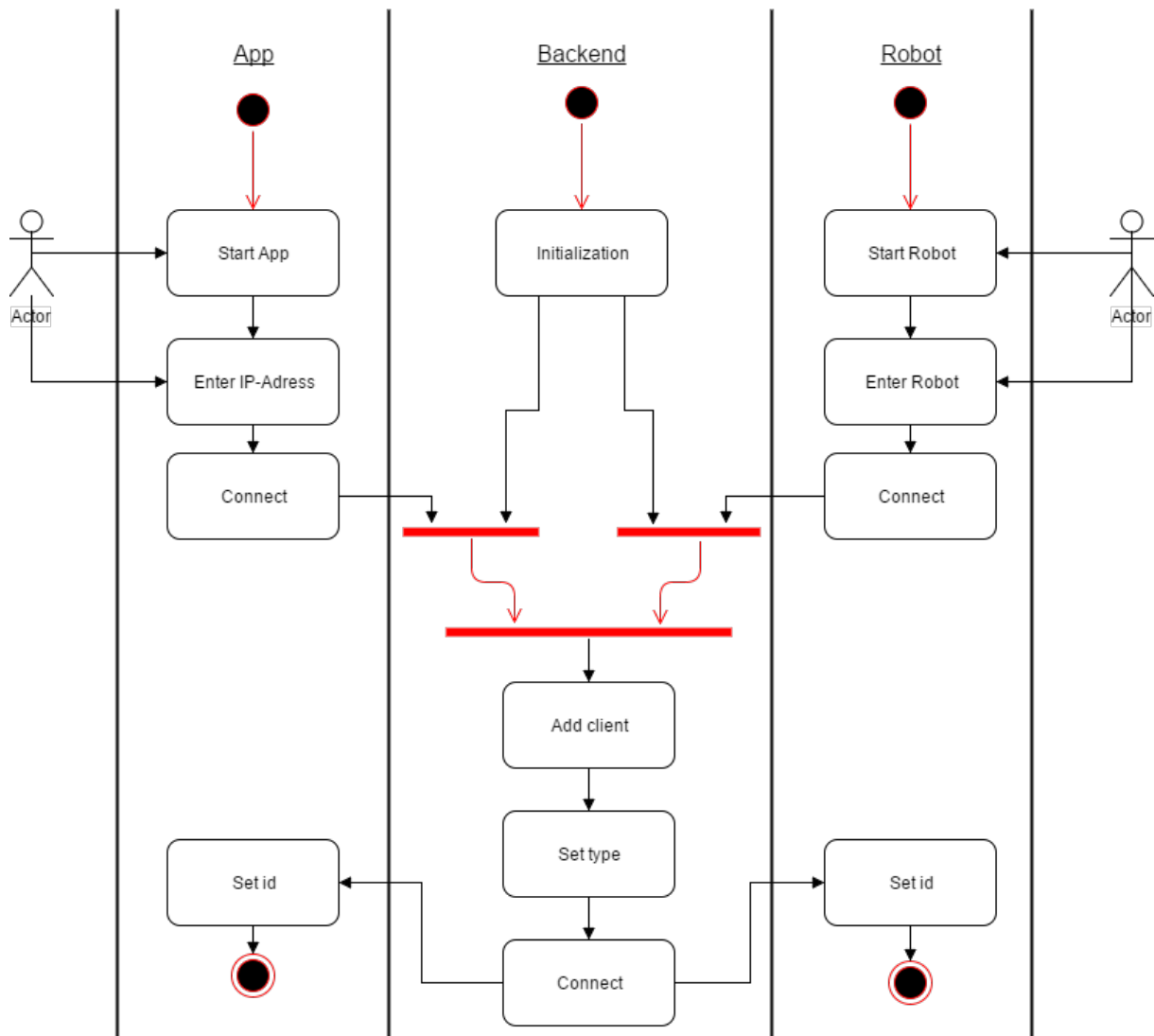


Abbildung 25: Use Case Connection



### 3.5.2 Synchronization

Der Use Case Synchronization stellt den Datenaustausch der beteiligten Komponenten dar, siehe Abbildung 27 und 28. 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, sowie eine verbesserte Steuerung schaffen.



Abbildung 26: Mockup Synchronization

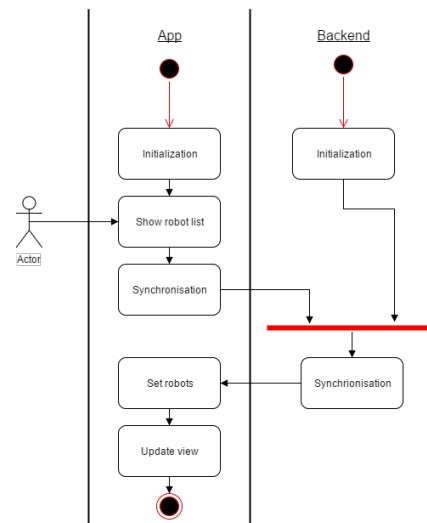


Abbildung 27: Use Case Synchronization

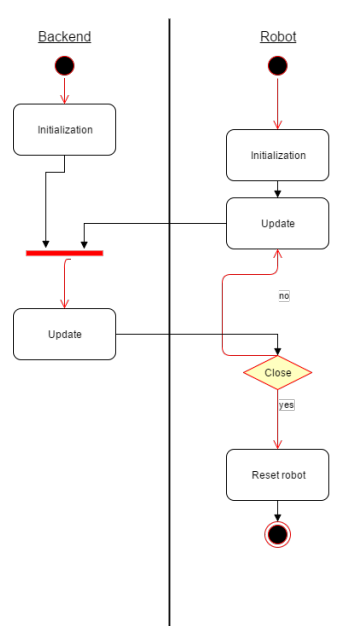


Abbildung 28: Use Case Update

### 3.5.3 Szenario

Der Use Case Szenario stellt den Ablauf der definierten Szenarien, zur Steuerung der Roboter dar, siehe Abbildung 30. 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.

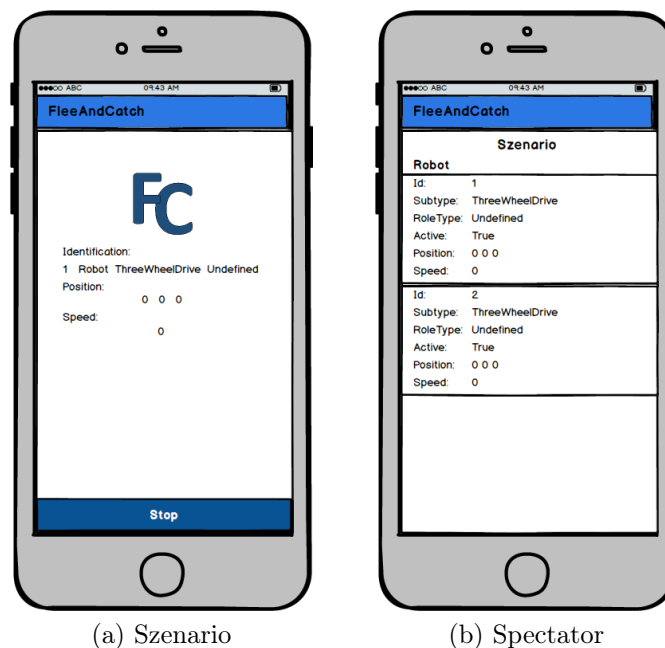


Abbildung 29: Mockup Szenario

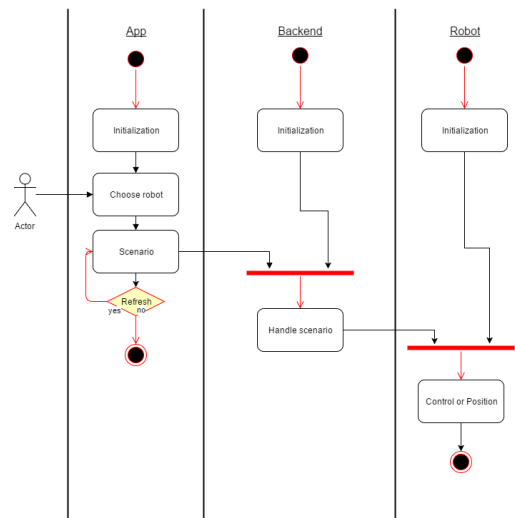


Abbildung 30: Use Case Szenario

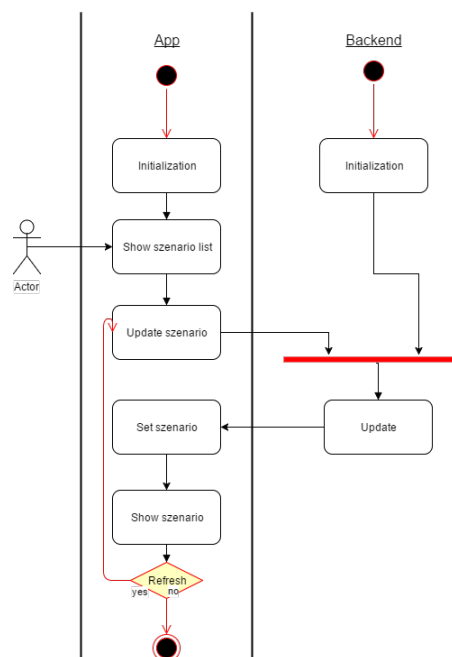


Abbildung 31: Use Case Spectator

### 3.5.4 Exception

Der Use Case Exception stellt den Ablauf einer auftretenden Exception in einem Roboter zum Kontext eines Verbindungsverlustes dar, siehe Abbildung 32. Dabei soll die auftretende Nachricht, sowie die beteiligte Komponente der Desktopanwendung zugesendet werden. Dies soll ein kontrolliertes Schließen eines Szenarios ermöglichen, welches von einer solchen Exception betroffen ist. Die dabei teilnehmenden Komponenten, wie Nutzer und andere Roboter sollen entsprechend zurückgesetzt werden, damit diese für ein neues Szenario zur Verfügung stehen.

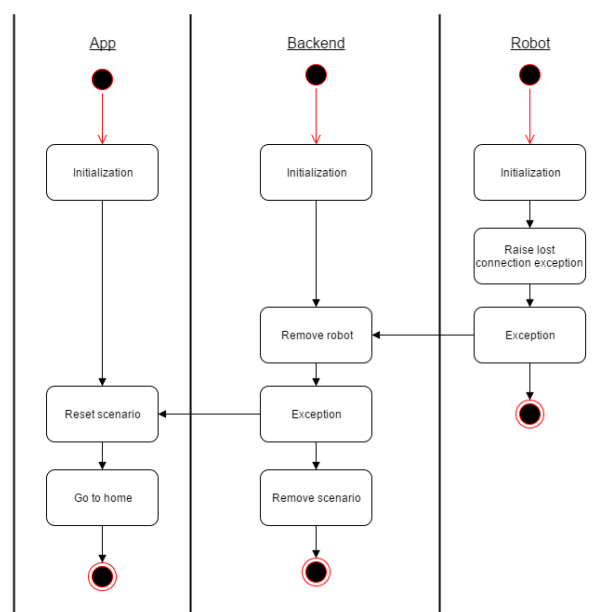


Abbildung 32: Use Case Exception

### 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 übertragenden Daten als Zeichenkette serialisiert und binär versendet werden, wobei die Gegenstelle die Daten über eine vorliegende Kommandostruktur deserialisiert. Um auf die Daten entsprechend zu reagieren werden diese nach Abhängigkeiten interpretiert. Dies soll über einen Typen realisiert werden, der der jeweiligen Struktur des Kommandos entspricht. Damit die Kommandostruktur als solche von den Komponenten von anderen 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	Stellt 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, wie 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	Stellt 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. Diese verändern sich je nach Typen 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 <b>API</b> 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 des auftretenden Fehlers, der behandelt wird. Verwendung findet die Fehlerweiterleitung im Verbindungsverlust des Roboters, damit andere Teilnehmer über das Abmelden des entsprechenden Roboters in Kenntnis gesetzt werden und das Szenario beenden.

Exception	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
Exception exception	Stellt die aktuelle Exception 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 <b>API</b> dar
ClientIdentification	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 abhängig des Roboters gesetzt wird.

Position	Definition
String id	Stellt den Identifikationsstring dar
String type	Stellte den Typen dar
String apiid	Stellt die verwendete <b>API</b> dar
ClientIdentification	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.

### 4.1 Kommunikation

Die Kommunikation der einzelnen Komponenten des Schwarmverhaltens baut auf einer klar definierten Struktur, um ein verteiltes System zu ermöglichen, siehe Abbildung 33. Die Daten werden dabei als JSON Objekte zur optimalen plattformübergreifenden Interpretation versendet, wobei jeweils die entsprechende Bibliothek zur Serialisierung verwendet wird.

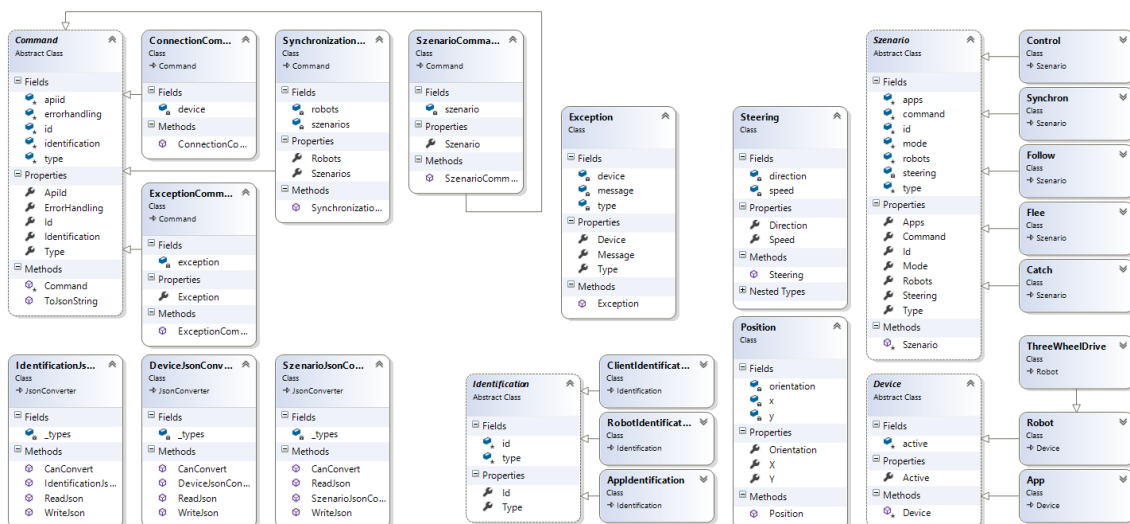


Abbildung 33: Aufbau Commands

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 34a. 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 34b.

```
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 34: Kommunikation

**Kommandos** stellen die Basis der Kommunikationsstruktur sowie den aktuellen Kontext dar, indem sich die Software befindet, siehe Abbildung 35. Sie enthalten grundlegende Attribute zur allgemeinen Identifikation des Kommandos, die zur Interpretation verwendet, welche über definierte Enums ausgewählt werden. Je nach Kommando sind zusätzliche Objekte enthalten, die durch die jeweilige Id vordefiniert sind.

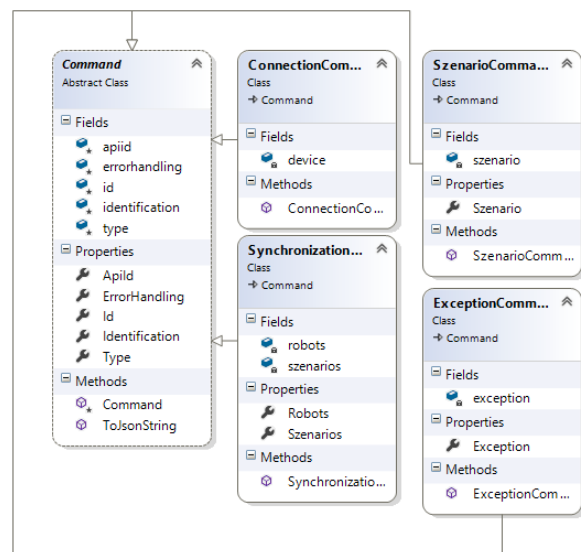


Abbildung 35: Kommandos

**Identifikationen** stellt die individuelle Identität der einzelnen Komponente dar, siehe Abbildung 36. Diese wird durch eine fortlaufende Identifikationsnummer, Typen und je nach Ableitung weiteren Attributen erreicht. Um die jeweiligen Kommandos entsprechend zuzuordnen, sind diese 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 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.

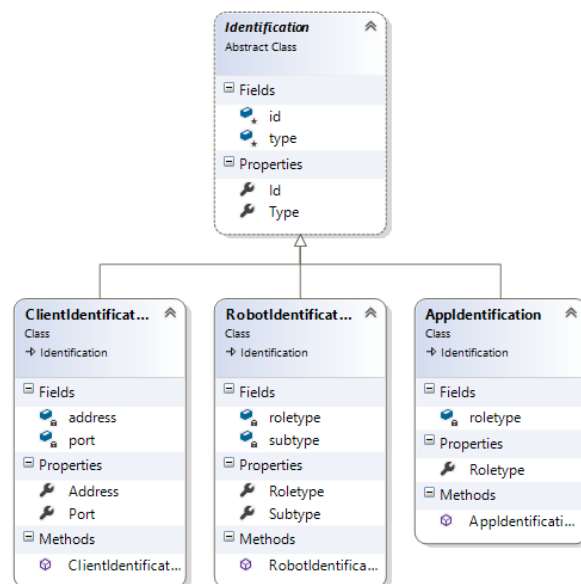


Abbildung 36: Identifikation

**Geräte** stellen die Komponenten dar, die an einem Szenario eines Schwarmverhaltens teilnehmen, siehe Abbildung 37. Sie enthalten jeweils spezifische Identifikations Objekte, zur gegenseitigen Zuordnung, sowie die erfassten Daten der entsprechenden Systeme. Die Unterscheidung erfolgt in zwei Komponenten, dem Robot und der App, wobei der Roboter in die jeweiligen Untertypen gegliedert werden kann.

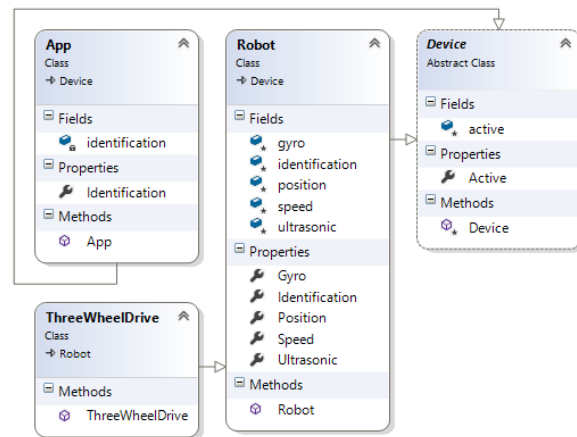


Abbildung 37: Devices

**Szenarios** stellen den Ablauf des Schwarmverhaltens dar, in dem sich der Nutzer befindet, siehe Abbildung 38. 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.

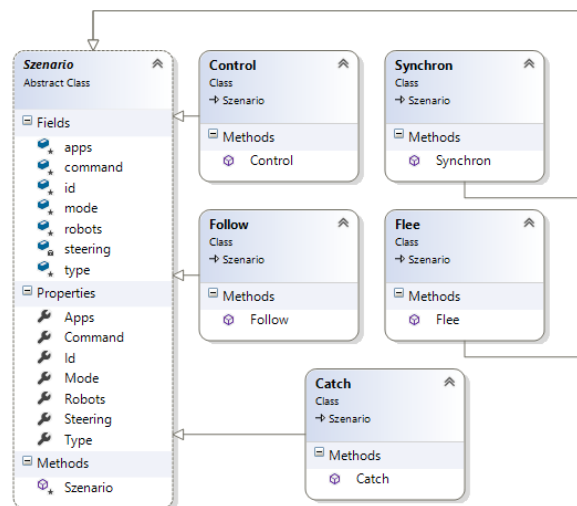


Abbildung 38: Szenarios

**Konverter** dienen der Deserialisierung von abstrakten JSON Objekten, welche nicht direkt identifiziert werden können, siehe Abbildung 39. Dazu gehören abstrakte Klassen, sowie Schnittstellen, welche keinem spezifischen Objekt zugeordnet werden kann. Die Implementierung erfolgt durch die Überschreibung der entsprechenden Methoden zur Deserialisierung und Serialisierung, siehe Abbildung 40a und 40b. Je nach Anwendung, wird ein Parameter übergeben, der das Objekt als Zeichenkette beinhaltet. Dieses wird durch eine Abfolge von Bedingungen auf den Typen geprüft wird, um das Objekt zu erstellen.

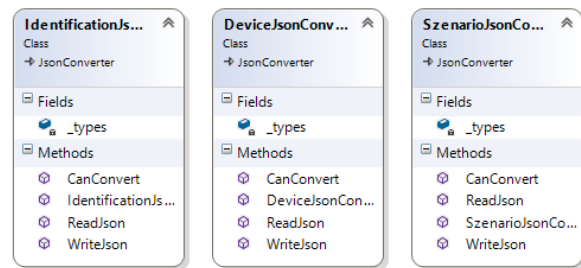


Abbildung 39: JsonConverter

```
3 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("Scenario 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;
        }
    }

    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 40: Device JsonConverter

## 4.2 App

### 4.2.1 Workflow

Die Struktur der **App** basiert auf dem in Xamarin weit verbreiteten ModelViewModelView Design Pattern, wobei die View durch Bindings aktualisiert werden kann.

**Model View ViewModel (MVVM)** ist ein Design Pattern, welches eine grundlegende Struktur in den Quellcode bringt. Dabei wird die erstellte Benutzeroberfläche von der Logik, sowie den Daten getrennt, um Änderungen unabhängig voneinander durchführen zu können. Um die Teile des Quellcodes zu verbinden wird auf Bindings gesetzt, die dafür zuständig sind, die jeweiligen Objekte gegenseitig zu aktualisieren.

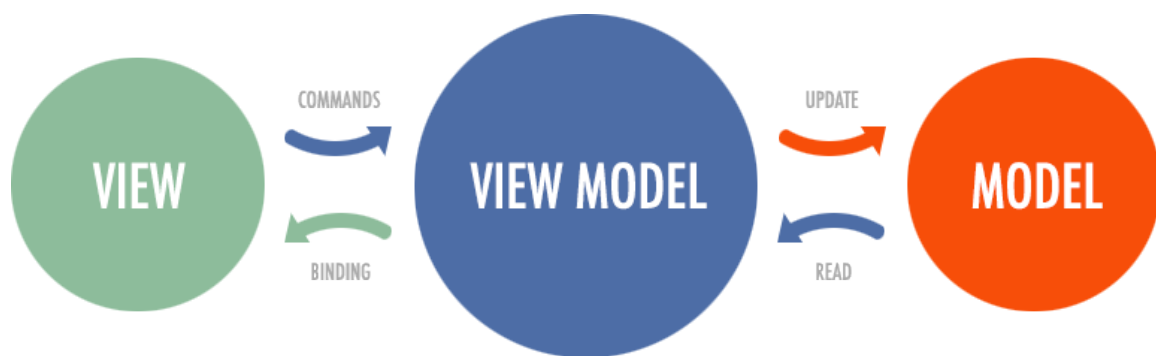


Abbildung 41: ModelViewModelView [? ]

**Bindings** sind für die gegenseitige Aktualisierung von Daten der jeweiligen Objekte zuständig. Sie entsprechen ihrem Sinn nach dem Observer Design Pattern und knüpfen an die ViewModels des MVVM an, wobei die dort definierten Objekte mit der gleichnamigen View und den Model Objekten verbunden sind. Durch eine Aktualisierung der Daten werden dabei die jeweilig verbundenen aktualisiert. Dies kann auf zwei verschiedenen Wegen von statten gehen. Manuell, durch das Aufrufen der entsprechenden Methode im ViewModel, andererseits automatisch, durch Attribute, wobei bei einer Änderung, die Methode automatisch angestoßen wird.

### 4.2.2 Graphical User Interface (GUI)

Die Oberfläche der **App** wird mittels XAML erstellt, welche auf der Basis von XML ist. Dabei werden die einzelnen Elemente untereinander strukturiert, wobei der Entwickler in Xamarin verschiedene Möglichkeiten des Designs besitzt. Grundlegend ist die Entscheidung, ob ein plattformspezifisches Design sinnvoll macht, oder aber ein plattformübergreifendes, wobei dieses weniger Freiheiten bietet. Das plattformübergreifende Design besitzt eine Reihe von grundlegenden Layouts, die für Xamarin eingesetzt werden

können, sowie unterschiedliche Seitenarten. Views Seiten (ContentPage, MasterDetailPage, NavigationPage, TabbedPage, TemplatedPage, CarouselPage) View (ContentPresenter, ContentView, ScrollView, Frame, TemplatedView) Layout (StackLayout, absolutLayout, RelativeLayout, GridLayout)

Weitere standardelemente

Durch verschiedene Packages können dabei viele Zusätzliche Dinge erstellt werden

**LogIn Page**

**Home Page**

**Option Page**

**List Page**

**Szenario Page**

**4.2.3 Buisnesslogic**

**4.2.4 Deployment**

**4.3 Backend**

**4.4 Robot**

## 5 Evaluation

- bluetooth
- delay
- struktur (kommunikation, design pattern)
- ev3 (Verhalten, installation, konfiguration)
- vergleich ziele, ergebnis



## 6 Ausblick

## Literatur

## Anhang