# Konzeption und Umsetzung eines Android-basierten Frameworks zur Ansteuerung von Nibo2-Anwendungen

## Bacheloarbeit

zur Erlangung des akademischen Grades

Bachelor

an der Technischen Hochschule Wildau


**Technische Hochschule Wildau**

Fachbereich: Ingenieurwesen/ Wirtschaftsingenieurwesen

Studiengang: Bachelor of Engineering (B. Eng.) Telematik

eingereicht von: Yerzhan Aitzhanov

eingereicht am: 13.04.2017

Gutachter/ innen: Prof. Dr. rer. nat. Mohnke, Janett,
BA Eng. Breßler, Jannine

# Antrag zur Bachelorarbeit

Technische Hochschule Wildau
Fachbereich Ingenieur- und Naturwissenschaften

Technische Hochschule Wildau
Technical University of Applied Sciences

## Bachelorarbeit

Eingang Antrag:

| | | | |
|---|---|---|---|
| Name, Vorname: | Yerzhan Aitzhanov | Matrikelnummer: | 100001445 |
| Studiengang: | Telematik | Seminargruppe: | T-13 |
| Betreuende/r Hochschuldozent/in: | Prof. Dr. rer. nat. Mohnke, Janett | Beginn der Arbeit: | 23. Januar 2017 |
| | | Abgabetermin: | 16. April 2017 |
| Zweitgutachter/in: | BA Eng. Breßler, Janine | Betriebliche Betreuer/in: | Prof. Dr. rer. nat. Mohnke, Janett |
| Themensteller: | Technische Hochschule Wildau | Straße: | Hochschulring 1 |
| | | PLZ, Ort: | 15745, Wildau |

© Technische Hochschule Wildau

**Kurzthema**

Konzeption und Umsetzung eines Android-basierten Frameworks zur Ansteuerung von Nibo2-Anwendungen

**Kurzthema in englisch**

Conception and development of an Android-based Framework for controlling of Nibo2 applications

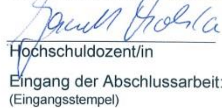
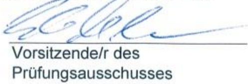
**Zielstellung**

Ziel dieser Bachelorarbeit ist es, eine Android-basierte Anwendung zur Verwaltung und Steuerung der Nibo2-Robotersystem-Anwendungen von Nicai Systems zu entwickeln. Dabei wird die Datenübertragung durch auf dem Roboter ausgestatteten Bluetoothmodul mit einem Android-basierten Tablet, an welchem das Framework läuft, erfolgen.

**Inhaltliche Anforderungen / Teilaufgaben**

- Definition der Anforderungen an das Framework zur Steuerung der Roboter-Anwendungen
- Konzepterstellung des Frameworks
- Implementierung eines entwickelten Konzeptes
- Testen des Gesamtsystems
- Erstellung der Anwender- und der Entwicklerdokumentation

Konsultationen erfolgen nach Vereinbarung mit dem betreuenden Hochschullehrer.

| Hochschuldozent/in | Vorsitzende/r des Prüfungsausschusses | Studierende/r |
|---|---|---|

Eingang der Abschlussarbeit: (Eingangsstempel)

Sperrvermerk ☐ Ja ☐ Nein

Fbl_3.4.5_06_05 Revisionsdatum: 11. 01. 2016 Seite 1 von 1

# Bibliographische Beschreibung

Autor: Aitzhanov, Yerzhan

**Konzeption und Umsetzung eines Android-basierten Frameworks zur Ansteuerung von Nibo2-Anwendungen.** [Bachelorarbeit, Technische Hochschule Wildau, 2017, 94 Seiten, 40 Abbildungen, 3 Tabellen, 4 Diagramme, 22 Quellenangaben, 11 Code Listings, 1 Beilage]

## Ziel:

Ziel dieser Bachelorarbeit ist es, ein *Android*-basierten Framework zur Ansteuerung des Roboterbausatzes *Nibo2* von *Nicai-Systems* zu konzipieren und zu entwickeln.

## Inhalt:

Anforderungsanalyse der zu entwickelnden Anwendung.

Konzeption und die Anwendungsmöglichkeiten des geplanten Frameworks.

Umsetzung des konzipierten Lösungsansatzes.

Ausblick auf die möglichen zukünftigen Weiterentwicklungen der Anwendung.

# Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Abschlussarbeit eigenständig angefertigt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

........................................ ......................................

(Ort, Datum) Yerzhan Aitzhanov

# Hinweise zum Lesen dieser Arbeit

Die in dieser Arbeit verwendeten Abkürzungen im Abkürzungsverzeichnis, Abbildungen im Abbildungsverzeichnis, Tabellen im Tabellenverzeichnis und verwendete Quellen im Quellenverzeichnis sind aufgeführt. Um auf Eigennamen, Fachbegriffe, Fremdwörter sowie Schlüsselbegriffe hinzuweisen, werden solche *kursiv* geschrieben und Erläuterungen zu wesentlichen Fachbegriffen im Glossar gegeben.

# Inhaltsverzeichnis

Antrag zur Bachelorarbeit ..... I

Bibliographische Beschreibung ..... II

Eidesstattliche Erklärung ..... III

Hinweise zum Lesen dieser Arbeit ..... IV

Inhaltsverzeichnis ..... V

1 Einleitung ..... 1

1.1 Zielsetzung ..... 2

1.2 Abgrenzung ..... 2

1.3 Ablauf der Arbeit ..... 3

2 Anforderungsanalyse ..... 4

3 Grundlagen ..... 7

3.1 Verwendete Hardware ..... 7

3.1.1 Nibo2 ..... 7

3.1.2 Blue-Modul ..... 11

3.1.3 Tablet ..... 11

3.2 Verwendete Software ..... 12

3.2.1 Android Studio ..... 12

3.2.2 VM Image ..... 12

4 Anwendungsfälle des Frameworks ..... 14

5 Konzept ..... 15

5.1 Systemarchitektur ..... 15

5.2 Datenaustausch ..... 17

5.2.1 Kommunikationsprotokoll ..... 17

5.2.2 Initialisierungsphase ........ 18
5.2.3 Systemprozesse der Anwendung ........ 20
5.2.4 Einfügen eines neuen *Nibo*-Projekts ........ 25
5.3 Storyboard ........ 29
6 Umsetzung ........ 35
6.1 *Nibo2*-Teilsystem ........ 35
6.1.1 Entstandene Probleme ........ 43
6.2 *Android*-Teilsystem ........ 47
6.2.1 Übersicht der wesentlichen Klassen ........ 47
6.2.2 Die *Activities* ........ 51
6.2.3 Entstandene Probleme ........ 59
7 Fazit und Ausblick ........ 66
7.1 Fazit ........ 66
7.2 Ausblick ........ 69
Abkürzungsverzeichnis ........ VII
Glossar ........ VIII
Quellenverzeichnis ........ XI
Abbildungsverzeichnis ........ XIV
Tabellenverzeichnis ........ XIX
Diagrammverzeichnis ........ XX
*Code Listings*-Verzeichnis ........ XXI
Beilagen ........ XXIII

# 1 Einleitung

Im Rahmen des Bachelorstudiums im 5. Semester des Studiengangs Telematik wird den Studierenden eine Reihe von verschiedenen Wahlpflichtfächern angeboten. Dazu gehört auch das Fach *C für eingebettete Systeme*. Dieses Fach stellt eine grundlegende Einführung für die Entwicklung der Anwendungssoftwarelösungen für Robotersysteme dar. Das Ziel ist dabei eine Vorstellung der Struktur von solchen Systemen am Beispiel des Roboterbausatzes *Nibo2* sowie die Sammlung von Basiskenntnissen im Umgang mit derartigen Anwendungssystemen zu bekommen.

Ein wichtiger Teilaspekt der Existenz der Technischen Hochschule Wildau ist es, neue Studierende für das Studium an dieser Bildungsanstalt zu begeistern. Um in diesem Teilaspekt konkurrenzfähig zu sein, werden die verschiedenen themengebundenen Veranstaltungen an der TH Wildau organisiert. Diese Veranstaltungen sind dafür geeignet, um den Abiturienten und den potenziellen Studenten über alle Vorteile bekannt zu machen, die im Laufe des Studiums von der TH Wildau angeboten werden. Der Telematik-Studiengang ist dabei auch keine Ausnahme. Eine solche Veranstaltung, wie der „*Tag der offenen Tür*“, dient als eine gute Möglichkeit, um bei den angehenden Programmierern für das Studium an der TH Wildau Interesse zu erwecken. Im Laufe dieser Veranstaltung wird der technische Reichtum des Telematik-Studiengangs vorgestellt. Dabei werden zum Beispiel die Roboterbausätze *Nibo2* und *NAO*-Roboter eingesetzt sowie für sie von den Telematik-Studenten entwickelte Anwendungen. Diese Anwendungen dienen dazu, das umfangreiche Möglichkeitsspektrum auf belustigende Art und Weise vorzustellen. Somit können den Veranstaltungsgästen anstatt von langen und umfangreichen informativen Vorträgen über die vorgestellten Roboter die Live-Demos der laufenden Roboter gezeigt werden.

In dieser Arbeit wird eine Beschreibung aller Entwicklungsprozesse, von der Konzeption bis zur Implementierung eines Werkzeugs gegeben, welches den Ablauf von solchen Veranstaltungen erleichtern könnte.

## 1.1 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung einer Lösung, die eine effiziente und in der Nutzung einfache Schnittstelle zur Ansteuerung der Robotersystemanwendungen darstellen soll.

## 1.2 Abgrenzung

Diese Arbeit wird durch die Entwicklung des Konzepts und seiner Implementierung für die Geräte unter dem *Android* Betriebssystem abgegrenzt. Eine Anwendungsentwicklung für andere Betriebssysteme ist nicht Gegenstand dieser Arbeit. Als Robotersystem wird nur der Roboterbausatz *Nibo2* betrachtet.

Die Realisierung dieses Projekts verfolgt keine Monetisierungsziele.

Die weiteren Ausnutzungsmöglichkeiten der fertigen Lösung grenzen sich auf keinen Fall durch die Verwendung im Rahmen der Veranstaltung „*Tag der offenen Tür*“ ab. Die möglichen Anwendungsfälle dieser Lösung werden in einem weiteren Kapitel konkreter beschrieben.

## 1.3 Ablauf der Arbeit

In dem Anfangskapitel *Anforderungsanalyse* werden alle aus der Zielstellung abgeleiteten Anforderungen an das zu entwickelnde System festgestellt. Neben den rein funktionalen Anforderungen werden auch die nichtfunktionalen Anforderungen formuliert.

Im Kapitel *Grundlagen* werden die für die Entwicklung von dem Framework verwendete Software- beziehungsweise Hardwarekomponenten beschrieben.

Im Kapitel *Konzept* wird die Architektur des ganzen Systems beschrieben. Dabei wird in die Untergliederung des Systems in die Bestanteile eingegangen. Für jeden Systembestandteil wird ein grober Entwurf mit zu erwartenden Ergebnissen vorgestellt.

Hieran schließt sich der praktische Teil der Arbeit im Kapitel *Umsetzung* an. Im abschließenden Kapitel erfolgen ein resümierendes Fazit zur vorliegenden Arbeit sowie ein Ausblick auf die zukünftige Entwicklung dieser Plattform.

# 2 Anforderungsanalyse

In diesem Kapitel werden alle Anforderungen an das zu entwickelnde Framework und auch an die *Nibo*-Anwendung definiert. Damit wird die Funktionalität des gesamten Systems konkretisiert. Für jede einzelne Anforderung wird ein bestimmter Typ:

- **F** – Funktionale Anforderung,
- **NF** – Nicht-Funktionale Anforderung,
- **Q** – Qualitätsanforderung,

sowie die bestimmte Priorität:

- **Muss** – Anforderung, die in dem Projekt realisiert werden muss,
- **Kann** – Anforderung, welche in dem Projekt realisiert werden kann, die aber keinen Hauptfokus in der Arbeit hat,

aufgeteilt.

Diese Anforderungen sind in den folgenden Tabellen 1, 2 und 3 aufgelistet und beschrieben:

| Nr. | Beschreibung | Typ | Priorität |
|---|---|---|---|
| **F1** | **Integration**<br>Ein Modul bestehend aus *.c und *.h-Datei, das die Anbindung eines existierenden *Nibo*-Projektes an das Framework realisiert. | **F** | **Muss** |
| **F2** | **Kommunikation (allgemein)**<br>Die Kommunikation zwischen *Android*-App und *Nibo*-Projekt erfolgt ausschließlich mit Hilfe von Bluetooth. | **F** | **Muss** |
| **F3** | **Kommunikation (*Nibo*-seitig)**<br>Das *Nibo*-Projekt ist in der Lage, sich softwareseitig mit dem mobilen Gerät zu verbinden, auf dem die | **F** | **Muss** |

| | | | |
|---|---|---|---|
| | *Android*-App läuft. | | |
| **F4** | **Kommunikation (*Android*-seitig)**<br>Die *Android*-App ist in der Lage, mit einem ausgewählten *Nibo*-Roboter, auf dem das *Nibo*-Projekt läuft, zu kommunizieren. | **F** | **Muss** |
| **F5** | **Broadcast**<br>Die *Android*-App kann parallel mit mindestens drei *Nibo*-Robotern kommunizieren, auf denen das *Nibo*-Projekt läuft. | **F** | **Muss** |
| **F6** | **StandBy-Modus**<br>Das *Nibo*-Projekt wartet nach der Verbindung auf ein Signal zum Start der entsprechenden Anwendung. | **F** | **Muss** |
| **F7** | **Rückmeldung**<br>Nach Beendigung der Anwendung schickt die *Nibo*-Anwendung eine Bestätigungsnachricht an die *Android*-App. | **F** | **Muss** |
| **F8** | **Dynamische Administration 1**<br>Die *Android*-App kann die *Nibo*-Roboter am System dynamisch an- und abmelden. | **F** | **Muss** |
| **F9** | **Dynamische Administration 2**<br>Von der Anmeldung aus kann die Anwendung auf jedem der *Nibo*-Roboter einzeln gestartet und auch gestoppt werden. | **F** | **Muss** |

**Tabelle 1: Funktionale Anforderungen**

| Nr. | Beschreibung | Typ | Priorität |
|---|---|---|---|
| **NF10** | **Leichte Bedienung**<br>Die *GUI*-Schnittstelle bietet für den Anwender eine intuitive und nachvollziehbare Menüführung. | **NF** | **Muss** |

**Tabelle 2: Nicht-Funktionale Anforderungen**

| Nr. | Beschreibung | Typ | Priorität |
|---|---|---|---|
| **Q11** | **Modularer Aufbau**<br>Das Framework ist modular aufgebaut, sodass ein Hinzufügen und Entfernen von *Nibo*-Projekten möglich ist. | **Q** | **Muss** |

**Tabelle 3: Qualitätsanforderungen**

# 3 Grundlagen

In diesem Kapitel werden alle verwendeten Hardwarekomponenten sowie die Softwarewerkzeuge aufgelistet und kurz beschrieben.

## 3.1 Verwendete Hardware

In diesem Unterkapitel wird eine Übersicht der eingesetzten Hardware gegeben.

### 3.1.1 Nibo2

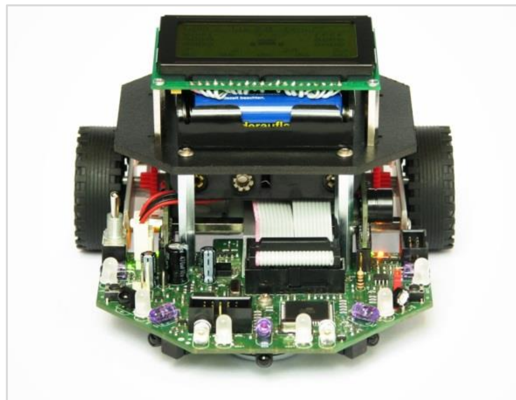


**Abbildung 1:** ***Nibo2***

Der Roboterbausatz *Nibo2* stellt einen kleinen programmierbaren Roboter dar, der im Rahmen des Faches *C für eingebettete Systeme* verwendet wird.

Der Roboter besitzt eine Menge von den eingebauten Bestandteilen, die die ganze Funktionalität gewährleisten. Dazu zählen:

- zwei Mikrokontroller, die als Kern dieses Systems zu betrachten sind:
  - ein Haupt-Controller *Atmel ATmega128*, der für solche in einem *Nibo2*-Roboter eingebauten Elemente, wie die Schweinwerfer, die Status *LED*s, die Liniensensoren, die Bodensensoren und die Versorgungsspannung, verantwortlich ist; [1]
  - ein Co-Controller *Atmel ATmega88*, der seinerseits für die Motorenansteuerung und Hinderniserkennung in der Umgebung zuständig ist; [2]
- zwei Motoren, die die eigentliche Bewegung des Roboters erledigen;
- eine Vielzahl von verschiedenen Sensoren, die eine Basisfunktionalität anbieten. Das sind:
  - die Bodensensoren: vier *CNY70* IR-Reflexlichtschranken, mittels derer die Bodenbeschaffenheit analysiert werden kann; [3]

**Abbildung 2: *CNY70*-Sensoren**

  - die Distanzmessungssensoren: mit Hilfe von IR-Fototransistoren und IR-LEDs ist eine grobe Entfernungsschätzung von Hindernissen möglich; [4]

**Abbildung 3: Distanzsensor**

- letztendlich die unterschiedlichen einsetzbaren Erweiterungsmodule, die das Spektrum der Roboterfunktionalität vergrößern, zum Beispiel:
    - ein Graphikdisplay, auf dem ein bestimmter Betriebs-Output eingeblendet werden kann; [5]

**Abbildung 4: Grafikdisplay**

    - ein *UCOM-IR2-X*-Programmieradapter, der speziell für diesen Roboterbausatz entwickelt wurde. Mittels dieses Programmieradapters können die eigenen geschriebenen Softwares auf dem Roboter installiert werden; [6]

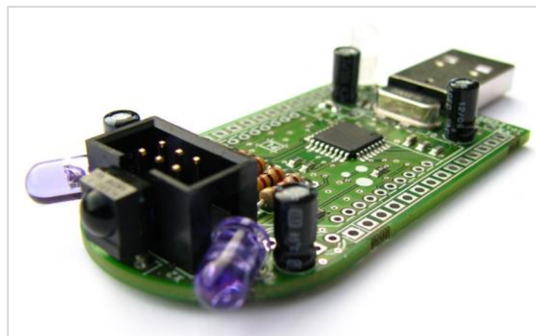


**Abbildung 5: UCOM-IR2-X**

    - ein *NDS3*-Erweiterungsmodul, das eine präzisere Distanzmessung im Vergleich zu den auf dem Roboter vorhandenen Distanzsensoren ermöglicht; [7]

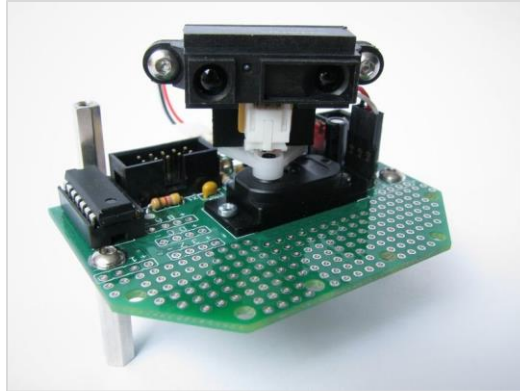


**Abbildung 6: NDS3- Erweiterungsmodul**

- eine *NXB2*-Adapterplatine, die eine Verwendung von einem *XBee*-Modul gewährleistet. [8]

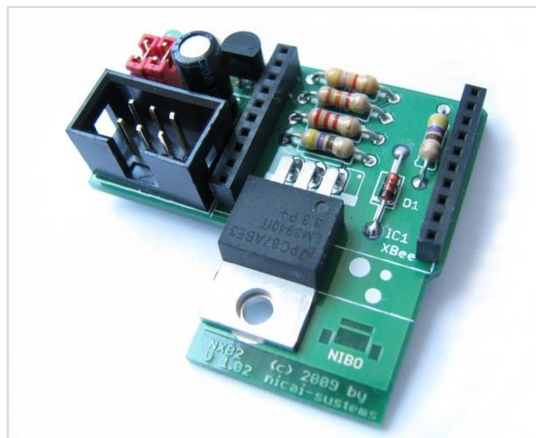



**Abbildung 7: NXB2-Adapterplatine**

Ein weiteres Modul, das eine besondere Rolle im Rahmen dieser Arbeit besitzt, ist ein Bluetooth-Modul. Eine Beschreibung dieses Moduls wird in dem folgenden Unterkapitel gegeben.

### 3.1.2 Blue-Modul

**Abbildung 8: Blue-Modul**

Mittels des *Blue Moduls* von *Nicai-Systems* ist es möglich per Bluetooth mit dem *Nibo2*-Robotersatz zu kommunizieren. Dabei erfolgt der Datenaustausch durch das *Nibo Serial Protocol*. [9]

### 3.1.3 Tablet

Da kein *Android*-basiertes Gerät bei dem Verfasser dieser Arbeit zur Verfügung stand, wurde entschieden ein Tablet aus der Hochschulbibliothek auszuleihen. Dieses wurde im Laufe der ganzen Implementierungsphase des geplanten Projektvorhabens als ein Werkzeug zum Testen verwendet.

**Abbildung 9: *Samsung Galaxy NOTE 10.1 SM-P600***

## 3.2 Verwendete Software

In diesem Unterkapitel werden alle bei der Implementierungsphase verwendeten softwaremäßigen Werkzeuge aufgelistet und kurz beschrieben.

### 3.2.1 Android Studio

*Android Studio* ist eine offizielle Entwicklungsumgebung von *Google I/O* für die Arbeit mit der *Android* Plattform. *Android Studio* basiert auf der plattformunabhängigen Entwicklungsumgebung *IntelliJ IDEA* von *JetBrains.* In *Android Studio* wird *Gradle* für alle *Build-Prozesse* des fertigen Anwendungsprogramms verwendet.

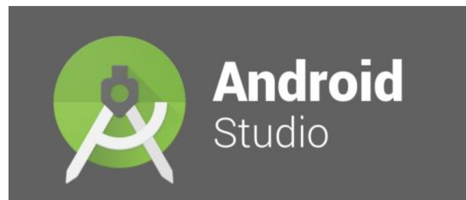



**Abbildung 10: *Android Studio* Logo**

### 3.2.2 VM Image

Eine *Virtuelle Maschine* ist eine softwaremäßige Implementierung eines physikalischen Rechners. Ein *VM* Image ist eine Kopie dieser *VM*, die ein bestimmtes Betriebssystem, die zu diesem Betriebssystem gehörige Anwendungen sowie die Anwendungsdaten beinhaltet.

**Abbildung 11:** ***VMWare Workstation*** **Logo**

Solche Images wurden als ein Bestandteil des Faches *C für eingebettete Systeme* für die Studierenden angeboten, um das spezifische Projektvorhaben im Rahmen dieses Faches zu realisieren.

Ein solches Image basiert auf dem Betriebssystem *Ubuntu* und besitzt eine Reihe von den Softwarewerkzeugen, die speziell für die Softwarelösungsentwicklung für den Roboterbausatz *Nibo2* geeignet sind.

Das im Rahmen dieser Bachelorarbeit ausgegebene VM Image besaß folgende Betriebsspezifikation:

- *Ubuntu Mate 15.04 64-Bit*

Darüber hinaus wurde die *VMWare Workstation* Plattform verwendet, um mit dieser VM umgehen zu können.

Der wesentliche Bestandteil der VM ist die Entwicklungsumgebung *Eclipse*, die bestimmte Einstellungen besitzt, um den Programmiervorgang eng auf den Mikrokontroller des *Nibo2*-Roboters zu spezifizieren.

Außerdem wurden die Softwarebibliothek zum Programmieren des *Nibo2*-Roboters in dieser Entwicklungsumgebung sowie die Beispiel-Projekte vorinstalliert. Jedes solche Beispiel-Projekt stellt ein simples Programm zur Ansteuerung eines ausgewählten Roboterbestandteils dar. Ein Beispiel wäre das Projekt „*HelloLEDStatus*“, das die auf dem Roboter eingebauten *LED*s in eine bestimmte Reihenfolge ein - beziehungsweise ausschaltet. Mittels dieser Beispiel-Projekte kann man eine Anfangsvorstellung entwickeln, welche weiteren Möglichkeiten im Einsatz mit den *Nibo2*-Robotern realisiert werden können.

# 4 Anwendungsfälle des Frameworks

Die Hauptanwendung dieses Frameworks ist die Vorstellung der im Studiengang der Fachrichtung Telematik angebotenen Ressourcen am Beispiel der Verwendung solcher Roboterbausätze im Rahmen der Veranstaltung „*Tag der offenen Tür*“.

Ein weiterer umfangreicher Anwendungsfall wäre die Verwendung dieses Frameworks als Basis für die Studierenden im Laufe der Lernprozesse. Dabei könnten die neuen Herausforderungen für die Weiterentwicklungen dieses Frameworks formuliert werden.

# 5 Konzept

In diesem Kapitel wird das Konzept für die Themenstellung dieser Arbeit beschrieben. Einschließlich soll ein Ansatz ausgearbeitet werden, mit dem dieses Konzept umgesetzt werden kann.

## 5.1 Systemarchitektur

Wie im Kapitel *Anforderungsanalyse* dargelegt wurde, gliedert sich das gesamte System konzeptionell in zwei Teilsysteme mit eigenen Anwendungskomponenten auf: *Nibo*-Teilsystem und *Android*-Teilsystem.

Die Systemarchitektur beschreibt den Aufbau und die Verknüpfung der einzelnen Anwendungskomponenten des jeweiligen Teilsystems.

In der folgenden Abbildung 12 ist die Systemarchitektur grafisch dargestellt. Diese Abbildung soll nur als eine grobe Vorstellung des zu entwickelnden Systems betrachtet werden. Es ist zu sehen, dass das jeweilige Teilsystem die benötigte Funktionalität besitzt, mit welcher die eigenen Teilaufgaben erledigt werden können. Es wird in den nächsten Unterkapiteln in die jeweilige Teilsystemarchitektur eingegangen und die konzeptionellen Ansätze näher beschrieben.

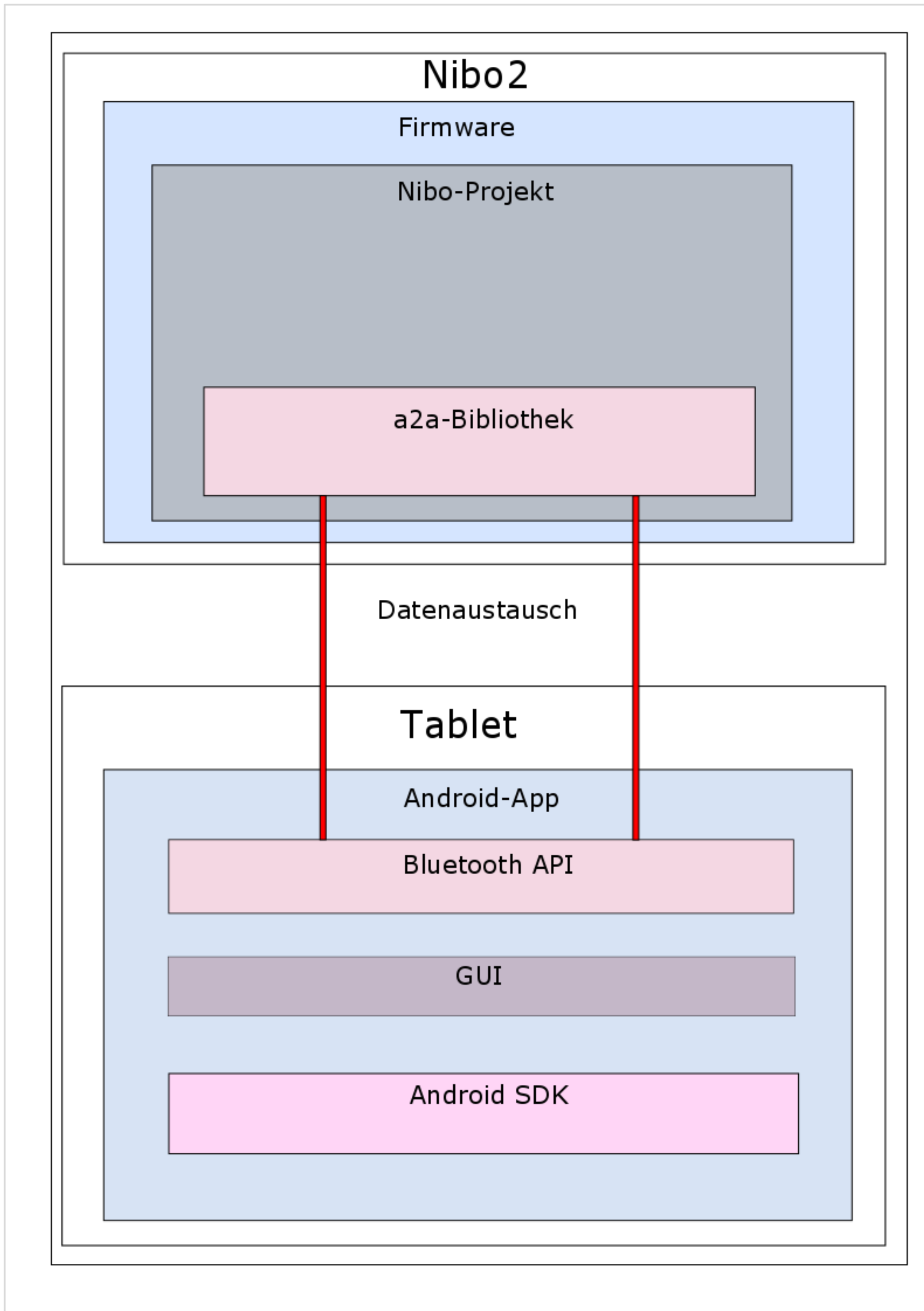



**Abbildung 12: Systemarchitektur**

## 5.2 Datenaustausch

Der Datenaustausch zwischen den Teilsystemen spielt in dem Projekt eine besondere Rolle. Dabei handelt es sich um einen effektiven Austausch von den wichtigsten Informationen bzw. Datensätzen von dem einen zu dem anderen Teilsystem. Davor müssen zwei wichtige Fragen beantwortet werden: welche Informationen sind relevant und wie werden diese gesendet?

### 5.2.1 Kommunikationsprotokoll

Die Lösung dafür ist die Entwicklung eines geeigneten Kommunikationsprotokolls und deren Einsatz im System. Mit Hilfe dieses Protokolls wird entschieden, welche Datensätze von Bedeutung sind und wie diese möglichst effizient zwischen den Teilsystemen generiert, gesendet und bearbeitet werden können.

Die Anforderung "**F2 Kommunikation (allgemein)**" aus dem Kapitel Anforderungsanalyse lautet, dass die Kommunikation zwischen *Android*-App und *Nibo*-Anwendung ausschließlich mit Hilfe von Bluetooth erfolgen soll. Dafür sind sowohl bei dem *Nibo*-Teilsystem, als auch bei dem *Android*-Teilsystem jeweils eigene spezielle Softwarebibliotheken vorhanden. Das ist einerseits die *UART1*-Bibliothek, welche den Datenaustausch über den seriellen *UART1* Port steuert, und anderseits die *Bluetooth API* der *Android SDK*, welche als Schnittstelle zur Steuerung der Bluetooth-Geräte dient.

Die Verwendung dieser Bibliotheken wird in dem weiteren Kapitel *Umsetzung* präziser beschrieben.

Im Folgenden wird der prinzipielle Ansatz, nach welchem das zu entwickelnde Kommunikationsprotokoll aufgebaut wird, näher betrachtet.

Jeder *Nibo*-Roboter wird ein eigenes *Nibo*-Projekt besitzen, das eine bestimmte Reihe an Aktionen ausführt. Als Beispiel für ein solches Projekt wäre die „freie Fahrt" zu erwähnen, in dem sich der *Nibo*-Roboter frei im Raum bewegt bzw. fährt und in der Lage ist, physikalische Hindernisse zu erkennen und diese zu umfahren. Dieses Projekt soll dann durch einen Knopfdruck von der *Android*-App aufrufbar sein. Durch das im *Nibo*-Roboter eingebaute Bluetooth-Modul und den seriellen Port *UART1* ist es möglich, einen Datenfluss zwischen den Teilsystemen aufzubauen. Dabei wird immer ein Byte als Dateneinheit gesendet. Genau dieses Merkmal wird als Basis zum Datenaustausch verwendet. Jedem bestimmten Anwendungssteuerbefehl wird ein ausgewähltes Symbol als Dateneinheit zugewiesen, wie z.B. ein Befehl zum Starten des Anwendungsbetriebs auf dem *Nibo*-Roboter.

### 5.2.2 Initialisierungsphase

Die weitere Herausforderung ist die Initialisierungsphase, bei welcher ein s.g. *hand shake* zwischen den beiden Teilsystemen erfolgt. Dabei soll diese Aktion möglichst dynamisch sein, um den Zeitverlust des gesamten Systems zu reduzieren. Es wurde folgender Ansatz zu der Kommunikation konzipiert:

1. Im Laufe der Initialisierungsphase sendet die *Nibo*-Anwendung bestimmte Datensätze zu der *Android*-App. Ein solcher Datensatz besteht dann aus 15 Bytes, wobei dem ersten Byte die vordefinierte ID eines Projekts zugewiesen wird. Den nächsten Bytes (max. 13) wird der Projektname zugewiesen und das letzte Endzeichenbyte steht für das Datenpaketende. In der Abbildung 13 ist ein solches Datenpaket schematisch grafisch dargestellt:

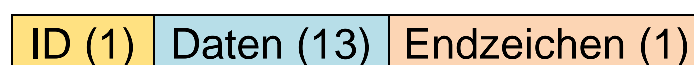



**Abbildung 13: Datenpaket für den Datenaustausch**

2. Die *Android*-App wird ihrerseits diese Datensätze empfangen, analysieren sowie die benötigten Informationsteile ausfiltern und diese weiterverwenden:

    a. Das erste Byte wird in eine separate Variable gespeichert und ist für das Senden eines Startsteuerbefehls eines *Nibo*-Projekts definiert.

    b. Aus den weiteren max. 13 Bytes wird der Projektname definiert und später auf der *GUI* der *Android*-App dargestellt.

    c. Das Endzeichen wird als Markierung verwendet, um das Ende eines Datenpakets zu kennzeichnen.

Wird zum Beispiel das Projekt „Freie Fahrt" auf dem *Nibo* initialisiert, so wird der String „1FREIE FAHRT#" von der *Nibo*-Anwendung generiert und zu der *Android*-App gesendet. Auf der Abbildung 14 ist dieser String grafisch dargestellt, wobei für diesen Projektnamen nur elf Bytes verwendet werden. Abhängig von der Projektnamengröße, die manuell in dem *Source Code* einzugeben ist, wird das Endzeichen „#" automatisch am Ende des Datenpakets eingesetzt. Ab diesem Punkt soll man zukünftig nur auf die Projektnamengröße achten, weil diese maximal 13 Bytes groß sein darf. In diesem konkreten Beispiel wird das Leerzeichen auch als ein Symbol mitgezählt.

| 1 | F | r | e | i | e |  | F | a | h | r | t | # |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

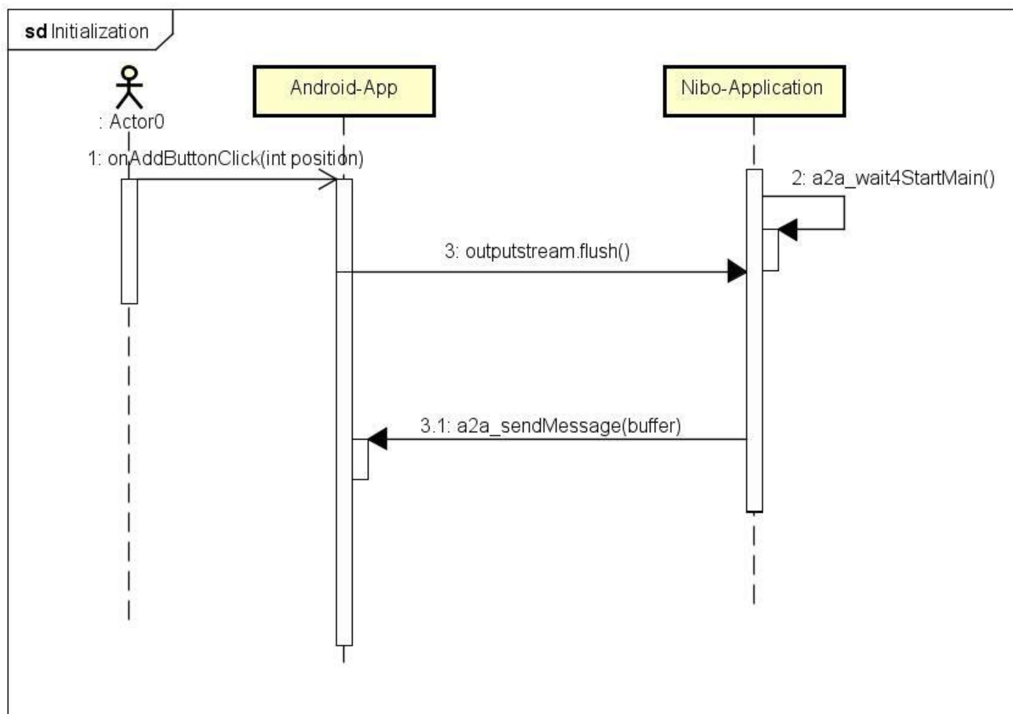
**Abbildung 14: Generiertes Datenpaket**

Die weiteren wichtigen Merkmale der Systemfunktionalität sind die Systemprozesse, die alle wesentlichen Systemtransaktionen, wie zum Beispiel Initialisierungsphase oder Anmelden einer *Nibo*-Anwendung an das System, steuern. Diese werden in dem folgenden Unterkapitel beschrieben.

### 5.2.3 Systemprozesse der Anwendung

Im Betriebsablauf der Hauptanwendung sollen alle wichtigen Prozesse systematisiert werden, um eine effiziente Systemfunktionsausnutzung zu gewährleisten. In den folgenden Sequenzdiagrammen 1 und 2 ist zu sehen, wie solche Systemprozesse erfolgen können.

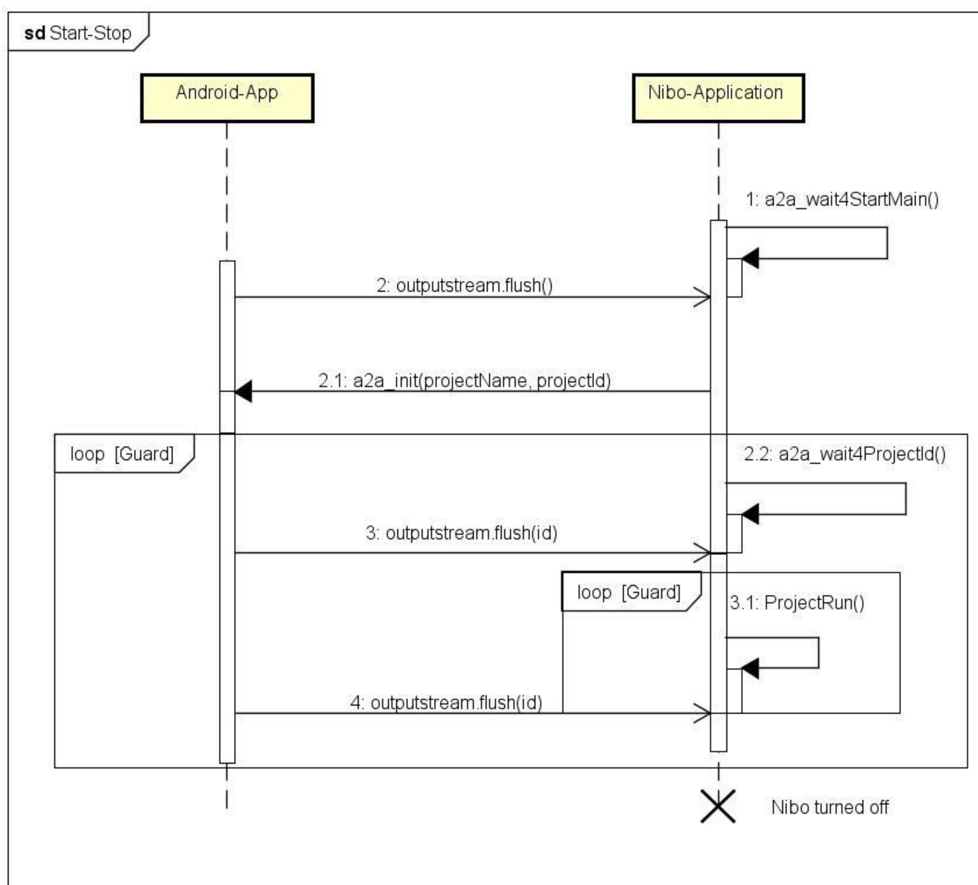
Das Sequenzdiagramm 1 beschreibt die Initialisierungsphase. Die *Nibo*-Anwendung verbleibt in einem Wartemodus, sobald sie einen bestimmten Steuerbefehl zum Starten des Betriebs bekommt. Ist dies der Fall, wird ein String mit der Information über das zur Verfügung stehende *Nibo*-Projekt generiert. Dabei werden in diesem String die vordefinierte Projekt-ID sowie der Projektname definiert. Abschließend wird dieser String zu der *Android*-App gesendet.




**Sequenzdiagramm 1: Initialisierungsphase**

Das Sequenzdiagramm 2 beschreibt einen weiteren Systemprozess für das Starten und das Stoppen der laufenden *Nibo*-Anwendung. Nach der Initialisierungsphase führt jedes Teilsystem folgende Aktionen aus:

- die *Android*-App bearbeitet das bekommene Datenpaket:
  - der Projektname wird auf der *GUI* eingeblendet
  - die Projekt-ID wird in eine separate Variable gespeichert und einem Knopfobjekt auf der grafischen Schnittstelle zugewiesen;
- die *Nibo*-Anwendung wartet bis ein Signal, bestehend aus der Projekt-ID, zum Starten sowie zum Stoppen des initialisierten Projekts vorkommt. Ist dies der Fall, wird das vorhandene Projekt von der *Nibo*-Anwendung ausgeführt. Sobald die Projekt-ID erneut übertragen wird, stoppt die Ausführung. Wird das Projekt gelöscht oder der *Nibo2*-Roboter, auf dem diese *Nibo*-Anwendung läuft, ausgeschaltet, wird das Projekt nicht mehr ausführbar.




**Sequenzdiagramm 2: Starten und Stoppen einer *Nibo*-Anwendung**

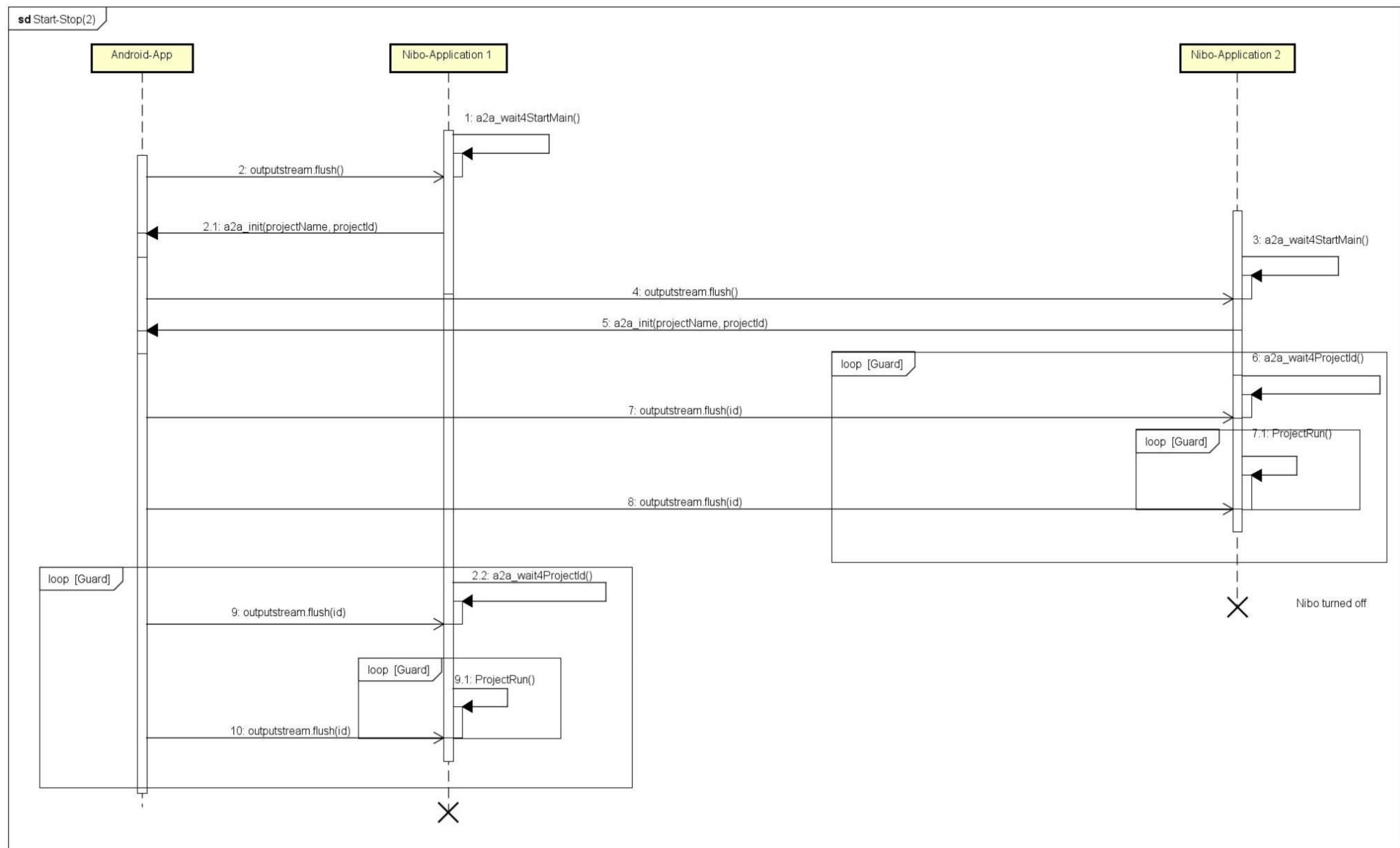
Die folgende Anmerkung ist zu beachten: bei der Transaktion *3.1:ProjectRun()* aus dem Sequenzdiagramm 2 wird der Ablauf der gesamten *Nibo*-Anwendung bezeichnet, die von diesem *Nibo*-Roboter ausgeführt wird. Da es technisch unmöglich war, mit einer Transaktion die umfangreiche *Nibo*-Anwendung zu kennzeichnen, wurde entschieden in diesem Fall diese Namenskonvention vorzunehmen.

Darüber hinaus gilt die erwähnte Konvention auch für das nächste Sequenzdiagramm 3. In diesem Sequenzdiagramm sind die in dem Sequenzdiagramm 2 beschriebenen Prozesse dargelegt, aber für den Ablauf mit zwei *Nibo*-Anwendungen.

Alle auf der Seite der *Android*-App gestarteten Prozesse sind asynchron und können damit beliebig zeitunabhängig initiiert werden. Dies ist aber auf der Seite der *Nibo*-Anwendung nicht der Fall. Alle Systemprozesse der *Nibo*-Anwendung sind auf die Prozesse von der *Android*-App synchronisiert. Ob der Wartemodus der *Nibo*-Anwendung im Moment läuft, die Initialisierungsphase angefangen hat, oder das Starten und Stoppen des initialisierten *Nibo*-Projekts verlaufen, alle diese Vorgänge sind streng von der Seite der *Android*-App ansteuerbar.

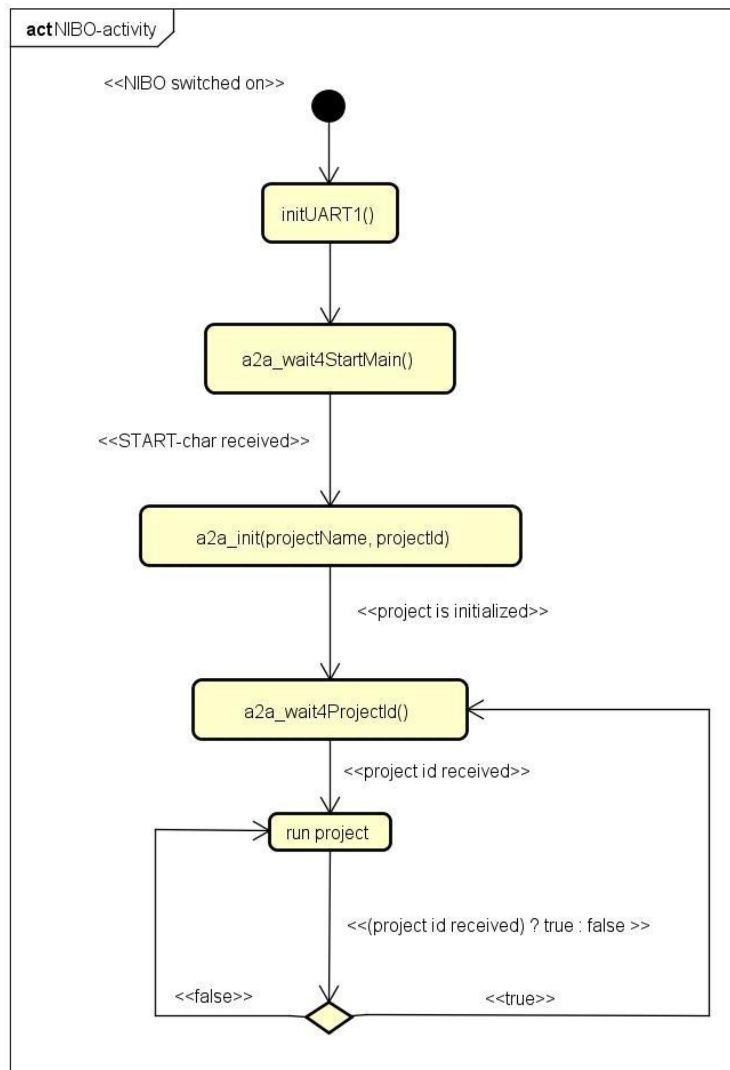
Aus den Sequenzdiagrammen 1, 2 sowie 3 sind alle bestimmten Systemprozesse durch die Pfeilspitzenart zu unterscheiden:

- die nicht ausgefüllte Pfeilspitzenart bezeichnet einen asynchronen Prozess
- die ausgefüllte Pfeilspitzenart bezeichnet einen synchronen Prozess




**Sequenzdiagramm 3: Starten und Stoppen zweier *Nibo*-Anwendungen**

Auf dem Aktivitätsdiagramm 1 sind die Inhalte der Sequenzdiagramme 1 und 2 zusammengefasst und grafisch dargestellt. Dabei ist zu erkennen, wie die wesentlichen Protokollaktionen mit dem vordefinierten *Nibo*-Projekt verbunden sind.




**Aktivitätsdiagramm 1: *Nibo*-Teilsystem**

Folgende Anmerkungen sind dabei zu beachten:

- die Aktivität mit dem Namen *run project* bezeichnet den Verlauf des eigentlichen vorhandenen *Nibo*-Projekts, da es technisch unmöglich ist, den umfangreichen Inhalt des *Nibo*-Projekts in einem Methodennamen zusammenzufassen;

- normalerweise sind in dem Kontext der Aktivitätsdiagramme sowohl die Startaktivität, als auch die Endaktivität vorhanden, da diese das Starten und das Stoppen des zu beschreibenden Systembetriebs kennzeichnen. Es wurde aber auf die Endaktivität in diesem Fall verzichtet, weil in dem Kontext des *Nibo*-Roboter-Systems diese Endaktivität durch das Ausschalten des Roboters gekennzeichnet wird und damit vor und nach jeweiliger gegebener Aktivität erfolgen kann, was im Endeffekt zu einer unübersichtlichen Darstellung des Diagramms führen könnte.

Abschließend ist ein weiterer Aspekt der Anwendung zu erwähnen. Dieser ist das Einfügen eines neuen *Nibo*-Projekts in das System. Dabei sollen alle neuen Projekte möglichst wenig programmatisch bei der Erweiterung geändert werden. Die dafür benötigten Schritte werden in dem folgenden Unterkapitel näher beschrieben.

### 5.2.4 Einfügen eines neuen *Nibo*-Projekts

In diesem Unterkapitel werden anhand eines Beispiels und den *Code Listings* alle wesentlichen Schritte zum Einfügen eines neuen *Nibo*-Projekts an das zu entwickelnde System beschrieben.

Dieses Beispiel stellt eine grobe Vorstellung des Lösungsansatzes für den erwähnten Vorgang des Einfügens eines neuen *Nibo*-Projekts dar. Die dafür verwendeten *Code Listing* Abschnitte sind als *Pseudo-Code* gegeben und sollen damit nur als der konzeptionelle Hintergrund betrachtet werden. Die dabei gegebenen Methodennamen könnten im Laufe der Implementierungsphase für die Optimierungszwecke umbenannt oder sogar gelöscht werden.

Die genauere Implementierungsbeschreibung findet man später im Kapitel *Umsetzung*.

Dabei wird ein vorhandenes *Nibo*-Projekt „Hinderniserkennung" als Beispiel angenommen, das die freie Fahrt eines *Nibo*-Roboters realisiert.

Die Abbildung 15 zeigt, wie die Ordnerstruktur von einem solchem Projekt aussehen könnte.

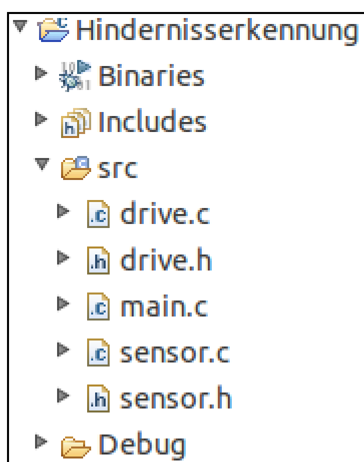



**Abbildung 15: Ordnerstruktur des vorhandenen Projekts**

Zuerst wird das vorhandene Projekt in die Entwicklungsumgebung kopiert, umbenannt und mit einem Ordner *adapt2Anroid* erweitert. Dieser Ordner soll die Quelldateien mit der Implementierung des Kommunikationsprotokolls durch das Bluetooth-Modul sowie die standardisierte Bibliothek zur Steuerung des seriellen Ports *UART1* enthalten, wie in der Abbildung 16 dargelegt ist.

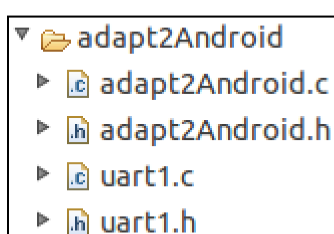



**Abbildung 16: Ordnerstruktur des A2A-Moduls**

Am Ende soll jedes neu einzufügende *Nibo*-Projekt die folgende Ordnerstruktur besitzen, wie es aus der Abbildung 17 zu sehen ist.

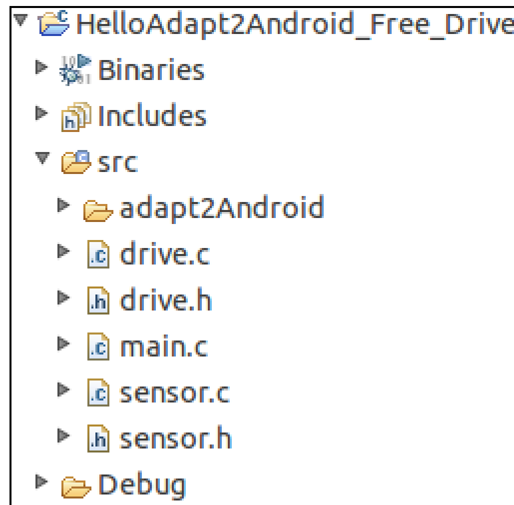



**Abbildung 17: Ordnerstruktur des fertigen Projekts**

Der nächste Schritt ist das Inkludieren der *a2a*-Bibliothek in die Quelldatei mit der *main*-Funktion: in diesem Fall ist es die Quelldatei *main.c*. Aus dem folgenden *Code Listing* 1 ist zu sehen, wie diese Bibliothek in das bestehende Projekt zu inkludieren ist. Dabei wird ein Verweis auf eine bestimmte Header-Datei definiert, die alle Funktionsprototypen, die die benötigten Konstanten sowie einen wichtigen Verweis auf die weitere Header-Datei *uart1.h* zur Verwendung des *UART1*-Ports beinhaltet. Diese Header-Datei wird standardmäßig als *adapt2Android.h* benannt.

```
#include "adapt2Android/adapt2Android.h"
```

**Code Listing 1: Inkludieren der Header-Datei**

Die weiteren Schritte befassen sich mit der Verwendung der Methoden zu dem *hand shake* mit der *Android*-App.

Als erstes kommt der Codeabschnitt, der einen Wartemodus realisiert. Dabei wird die *Nibo*-Anwendung in diesem Modus verbleiben, bis ein spezifisches vordefiniertes Symbol in dem Empfangspuffer von der *Android*-App vorkommt. Ist dies der Fall, erfolgt die bekannte Initialisierungsphase durch den Aufruf der Methode *a2a_init()*. Dabei wird ein String aus den manuell eingegebenen Parametern, welche den Projektnamen und die Projekt-ID umfassen, generiert

und in die *Android*-App gesendet. Aus der *Code Listing* 2 ist zu sehen, wie die beschriebenen Aktionen programmatisch realisiert werden könnten.

```
//wait until START byte is received
a2a_wait4StartMain(input);

//initialize the local project;
a2a_init(projectName, projectId);
```

**Code Listing 2: Standby-Modus und die Initialisierung des Projekts**

Der *Code Listing* 3 zeigt, wie der letzte Schritt zum Einfügen des neuen Projekts in das System erfolgt. Nach der Initialisierungsphase wird die Anwendung noch in einem Wartemodus verbleiben, bis der Empfangspuffer eine Nachricht mit der Projekt-ID von der *Android*-App bekommt. Die Ausführung des vorhandenen *Nibo*-Projekts wird erfolgen, sobald ein Signal zum Stoppen erhalten wird.

```
//operational loop
while(1){

      //wait until project id is received
      a2a_wait4StartProject(input);

      while(a2a_wait4StopProject(input){

            //run the actual project

      }

}
```

**Code Listing 3: Starten und Stoppen des vorhandenen Projekts**

Auf alle erwähnten Methoden zum Kommunikationsaufbau, beziehungsweise zum Umgang mit der *hand shake* Tätigkeit wird in dem Kapitel *Umsetzung* näher eingegangen und an dieser Stelle präziser beschrieben und erläutert.

## 5.3 Storyboard

Als ein *Storyboard* bezeichnet man eine grafische Darstellung der *GUI* einer mobilen Applikation, mit allen Applikationsfenstern und ihren Inhalten sowie mit allen logischen Verknüpfungen von diesen Applikationsfenstern.

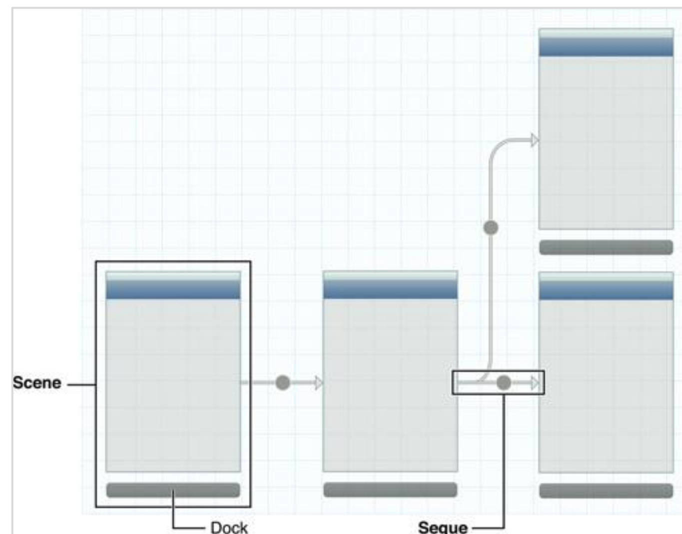



**Abbildung 18: *Storyboard* – Beispiel**

Damit ist ein App-Entwickler in der Lage, einen groben Entwurf des zu entwickelnden Systems zu erstellen. Dabei werden die wesentlichen Funktionalitäten zusammengefasst. Zu jedem Zeitpunkt im Laufe der Implementierungsphase bekommt der Entwickler eine Übersicht der entworfenen sowie geplanten Funktionsausstattung der Applikation. Darüber hinaus ist dieses Werkzeug als ein Gerüst hilfreich, auf dem sich für alle erstellten Prototypen zukünftig funktionsfähige Methoden ableiten lassen.

Im Folgenden wird ein solches *Storyboard* für das geplante System gegeben. Damit soll eine grobe Vorstellung der zu entwickelnden App-Funktionalität vermittelt werden. Es wird auf alle Applikationsfenster eingegangen und ihre Inhalte sowie die Hauptfunktionalitäten beschrieben.

Als erstes kommt das Hauptapplikationsfenster (*Main Activity*) vor. Auf der Abbildung 19 ist dieses Fenster grafisch dargelegt und es ist zu sehen, welche grundlegende *GUI* Elemente dabei vorhanden sein sollen.

**Abbildung 19: *Storyboard* – Main Activity**

Dies umfasst drei Knöpfe, die bestimmte Systemfunktionen ausführen:

- *ADD PROJECT* Knopf, der einen Verweis auf das weitere Applikationsfenster enthält, das für die Suchfunktion von allen Bluetooth Geräten zuständig ist. Nach dem Knopfdruck soll diese Suchfunktion gestartet werden, das Bluetooth-Medium soll gescannt werden und am Ende der Suchfunktion soll die App auf das weitere Funktionsfenster weiterleiten, wo eine Liste der gefundenen Bluetooth-Geräte eingeblendet werden soll;
- *DELETE PROJECT* Knopf, der einen Verweis auf das weitere Applikationsfenster enthält, das für das Löschen der im App-System initialisierten *Nibo*-Anwendungen verantwortlich ist. Nach dem Knopfdruck leitet die App in das weitere Fenster weiter, wo alle

vorhandenen *Nibo*-Anwendungen aufgelistet werden sowie die Knöpfe zum eigentlichen Löschen des Projekts aus dem System.

- *RUN PROJECT* Knopf, der einen Verweis auf das weitere Applikationsfenster enthält, das das Starten und das Stoppen der im App-System initialisierten *Nibo*-Projekte ansteuert. Nach dem Knopfdruck wird in das spezifische Fenster weitergeleitet, wo alle vorhandenen *Nibo*-Anwendungen aufgelistet werden sowie die Knöpfe zum eigentlichen Starten und Stoppen der *Nibo*-Anwendung.

Aus der weiteren Abbildung 20 ist zu erkennen, welche wesentlichen Elemente der grafischen Oberfläche das weitere Applikationsfenster *Device List Activity* beinhalten soll.

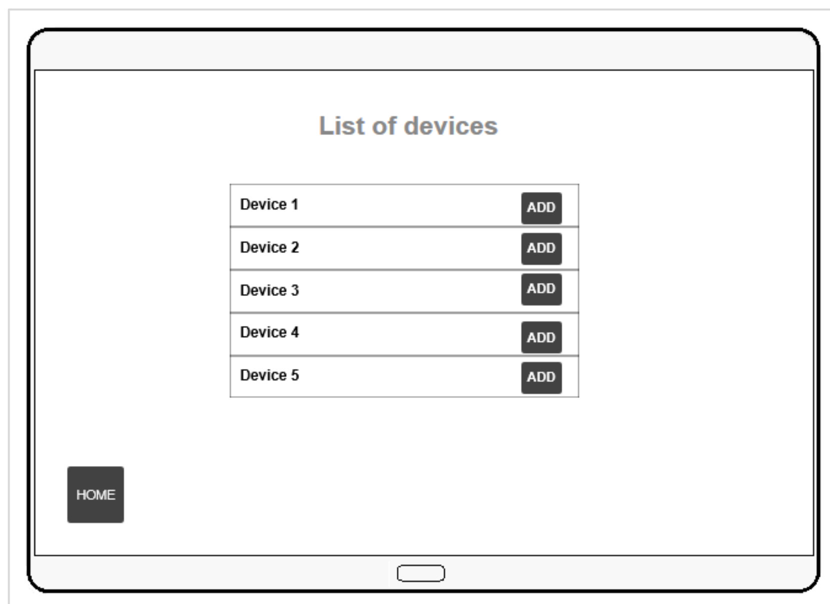



**Abbildung 20: *Storyboard* – Device List Activity**

Folgende wichtigste Elemente sollen dabei vorhanden sein:

- eine Tabelle, die eine Reihe von weiteren Elementen pro Zeile zur Verfügung stellen soll:

  - die Information, wie zum Beispiel der Name und die *Mac*-Adresse, der gefundenen Bluetooth-Geräte nach dem Suchvorgang
  - ein Knopf zum Initiieren des Initialisierungsvorganges von einer *Nibo*-Anwendung
- ein Button zum Zurückkehren in das Hauptfenster.

Der grafische Entwurf des nächsten Funktionsfensters *Run Projects Activity* ist in der Abbildung 21 zu sehen.

Die strukturelle Funktionsfensterausstattung ist ähnlich, wie bei dem *Device List Activity* – bestehend aus einer Tabelle mit einem bestimmten Inhalt und einem Knopf zum Zurückkehren ins Hauptfenster.

Der einzige Unterschied zwischen diesen beiden Fenstern liegt aber in den Inhalten der Tabellen. Auf der Tabelle des *Run Projects Activity* Funktionsfensters werden pro Zeile der Projektname der initialisierten *Nibo*-Anwendungen eingeblendet sowie ein Knopf zum Ansteuern des Start- beziehungsweise Stoppvorgangs dieser *Nibo*-Anwendung.

**Abbildung 21: *Storyboard – Run Projects Activity***

Das letzte Funktionsfenster *Delete Projects Activity* ist für den Löschvorgang des vorhandenen *Nibo*-Projekts zuständig. Dieses Funktionsfenster ist in der Abbildung 22 grafisch dargelegt.

Dieses Funktionsfenster ähnelt dem *Run Projects Activity* Funktionsfenster sowohl strukturell als auch inhaltlich. Sie unterscheiden sich voneinander nur durch den Knopfnamen sowie der kommenden Aktion nach dem Knopfdruck der jeweiligen Tabellenzeile. In diesem Fall wird das vorhandene *Nibo*-Projekt, wie der Name des Knopfs lautet, aus der App gelöscht und wird ab diesem Moment nicht mehr von der App aufrufbar sein.

**Abbildung 22: *Storyboard – Delete Projects Activity***

Auf der abschließenden Abbildung 23 sind alle beschriebenen Funktionsfenster der App systematisch zusammengefasst und grafisch dargestellt. Die dabei dargelegten Pfeile mit zwei Pfeilspitzen bezeichnen die bidirektionalen Beziehungen zwischen den Funktionsfenstern.

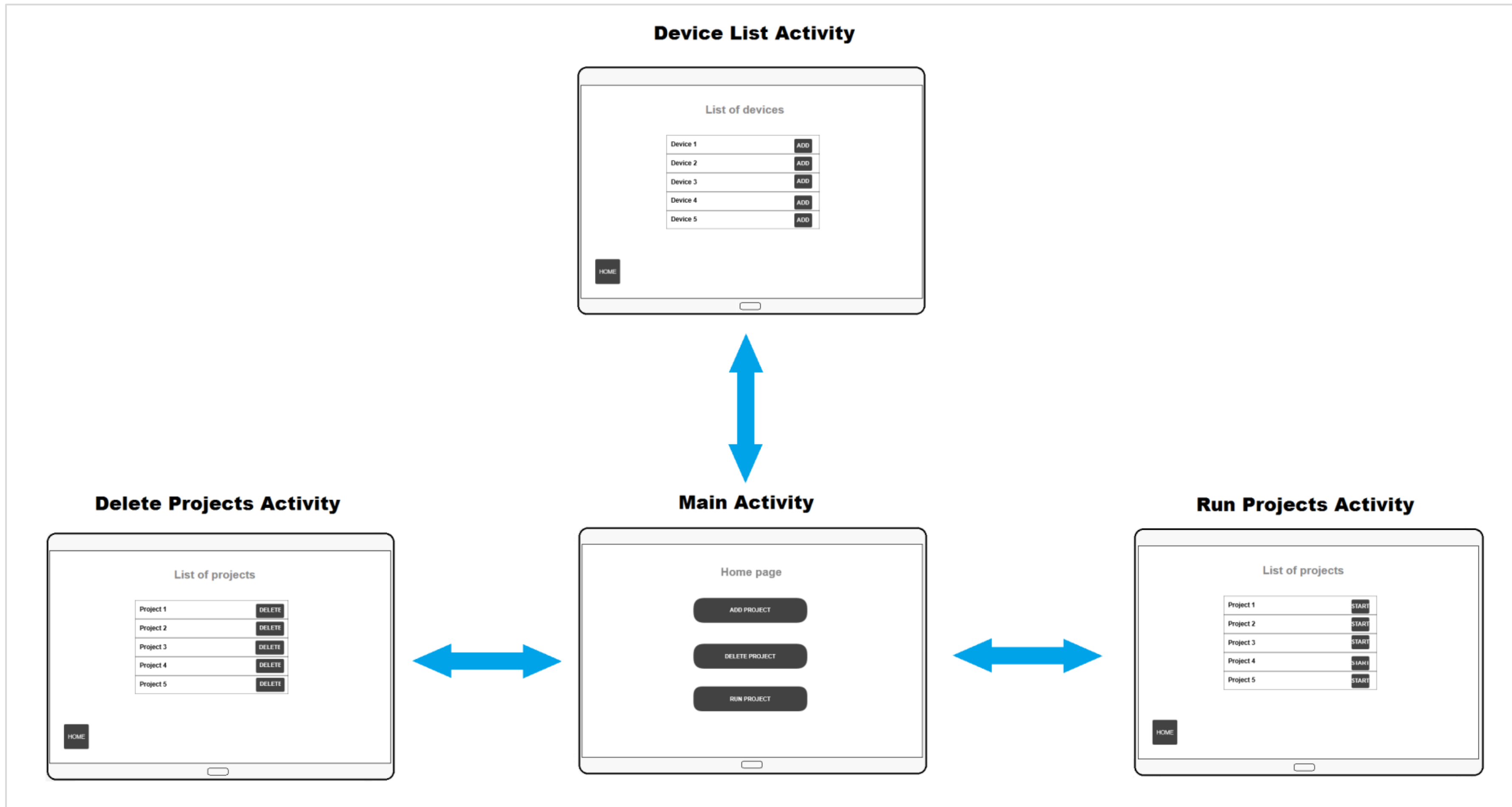



**Abbildung 23:** ***Storyboard*** **– Zusammenfassung**

# 6 Umsetzung

In diesem Kapitel wird die präzisere Beschreibung des im Rahmen dieser Arbeit entworfenen Konzeptes gegeben. Wie bei dem Kapitel *Konzept* dargelegt wurde, gliedert sich dieses Kapitel auch hauptsächlich in zwei Unterkapitel: *Nibo2*-Teilsystem und *Android*-Teilsystem. Dabei bezeichnet jedes Unterkapitel das jeweilige Teilsystem.

Der ganze Ablauf des kommenden Kapitels wird folgendermaßen erfolgen:

- als erstes wird der Endlösungsansatz vorgestellt und deren Hauptfunktionalität konkreter beschrieben;
- abschließend werden in einem weiteren Unterkapitel *Entstandene Probleme* die im Laufe der ganzen Implementierungsphase entstandenen Grundprobleme erwähnt und beschrieben. Genau mittels dieses Unterkapitels kann man den ganzen Entwicklungsweg anschaulich verfolgen. Die dabei entstandenen Problemlösungen hatten einen direkten Einfluss auf die Herausbildung des Endlösungsansatzes.

## 6.1 *Nibo2*-Teilsystem

In diesem Unterkapitel wird ein Endlösungsansatz des *Nibo2*-Teilsystems gegeben und auf die Hauptaspekte der Implementierungsphase eingegangen.

Die Ordnerstruktur aus der Abbildung 16, die in dem Kapitel *Konzept* dargelegt wurde, wurde im Laufe der ganzen Implementierungsphase als ein Gerüst verwendet und die *Source*-Datei *adapt2Android.c* sowie die Header-Datei *adapt2Android.h* folgendermaßen gefüllt:

- die *Header*-Datei beinhaltet die Prototypen aller zu entwickelnden Methoden;
- die *Source*-Datei beinhaltet die Implementierungen dieser in der Header-Datei erwähnten Prototypen.

Im Laufe der Implementierungsphase wurde eine Methodennamenkonvention vorgenommen, um die Gleichheitsfälle von dem Methodennamen bei der Zusammenfassung mit anderen *Nibo*-Anwendungen zu vermeiden. Dabei wurde jedem Methodenname das Präfix *„a2a_“* hinzugefügt.

Das ganze Teilsystem *Nibo2* basiert hauptsächlich auf drei Methoden, die die Hauptfunktionalität gewährleisten. Das sind:

- `bool a2a_init(char[] data, char id),`
- `bool a2a_wait4StartMain(),`
- `bool a2a_wait4ProjectId().`

Außerdem sind folgende Zusatzmethoden vorhanden, die weniger einen Einfluss auf die gesamte Teilsystemfunktonalität besitzen, aber in sich eine notwendige Hilfestellung für die drei Hauptmethoden gewährleisten. Das sind:

- `void a2a_sendChar(char *data),`
- `void a2a_sendMessage(char []data),`
- `void a2a_strLength(char *s).`

Im Folgenden werden anschließend die drei Hauptmethoden anhand der Quellcodeangabe näher beschrieben, die Hilfsmethoden kurz beschrieben und abschließend mittels eines kleinen Beispiels die Verwendung der entwickelten Anwendung gegeben.

### `a2a_init()`- Methode

Die Methode steuert den bekannten Initialisierungsvorgang von einem *Nibo*-Projekt in der *Android*-App. Damit sind als Input die zwei Parameter einzugeben: der gewünschte Projektname, der letztendlich auf der *GUI* der

*Android*-App eingeblendet wird und die spezifische Projekt-ID, die das zu initialisierende *Nibo*-Projekt im System kennzeichnet.

```
void a2a_init(char data[], char id) {
	//save the project id
	projectId = id;
	//define the payload buffer
	char buffer[A2A_BUFFERSIZE];
	//clear buffer
	buffer[0] = 0x00;
	//set first byte of payload
	buffer[0] = id;
	//set the data
	for (int i = 0; i < a2a_strLength(data); i++) {
		buffer[i + 1] = data[i];
	}
	//set the last byte of payload with delimiter
	buffer[a2a_strLength(data) + 1] = A2A_PAYLOAD_END;
	//send payload
	a2a_sendMessage(buffer);
}
```

**Code Listing 4: `a2a_init()`- Methode**

Zuerst werden die wichtigen Variablen definiert beziehungsweise initialisiert. Einerseits handelt es sich hierbei um eine globale Variable innerhalb dieser Quelldatei zur Speicherung der Projekt-ID und anderseits um eine lokale Variable, die als ein Datenpuffer aus einem *Array* von Zeichen für den Projektnamen dient. Danach wird dieser Puffer gemäß der *C*-Programmiersprachenspezifikation durch den Einsatz des ersten Elements des Zeichen-Arrays mit „0x00“ geleert. Nach dem Leeren des Arrays erfolgt das Auffüllen des vorhandenen Puffers mit den Daten. Dabei werden das erste Byte des Puffers auf die Projekt-ID und der Pufferrest auf den Projektnamen gesetzt. Das Auffüllen des Puffers mit dem Projektnamen erfolgt folgendermaßen:

mittels einer `for`-Schleife wird jedes Element des Puffers iteriert und jedes Zeichen des Projektnamens eingesetzt. Als nächstes wird das Ende des Pakets definiert. Dies erfolgt durch den in dem Kapitel *Konzept* dargelegten Mechanismus, bei welchem durch den Einsatz des Symbols „#" das Paketende ermittelt wird. Dieses Symbol wird explizit in der Konstante `A2A_PAYLOAD_END` gespeichert. Darüber hinaus soll der zukünftige Anwender auf die Projektnamengröße aufpassen: diese soll maximal 13 Bytes betragen, da die gesamte Puffergröße gleich 15 Bytes umfasst und zwei Bytes für die Projekt-ID und das Endsymbol reserviert sind. Abschließend wird das generierte Paket mittels der Hilfsmethode `a2a_sendMessage()` durch die *UART1*-Schnittstelle an die *Android*-App gesendet.

### `a2a_wait4StartMain()`- und `a2a_wait4ProjectId()`- Methoden

Hauptsächlich gewährleisten die beiden Methoden die folgende Funktionalität: sie prüfen den Empfangspuffer der *UART1*-Schnittstelle, ob ein Byte mit einem bestimmten Zeichen vorkommt. Ist dies der Fall, liefert die jeweilige Methode die `true` Ausgabe.

Zuerst wird eine lokale boolesche Variable `received` definiert beziehungsweise initialisiert und auf `false` gesetzt. Danach werden die Methoden mittels des `if`-Statements und der Methode `uart1_rxempty()` in einem Wartemodus verbleiben, bis der jeweilige Empfangspuffer nicht mehr leer ist. Ist dies der Fall, werden die Inhalte der Empfangspuffer geprüft, ob sie eine von den zu erwartenden Eingaben beinhalten:

- bei der Methode `a2a_wait4StartMain()` geht es um ein vordefiniertes Symbol „+", das in der globalen Konstante `A2A_START_ACTION` gespeichert wurde
- und bei der Methode `a2a_wait4ProjectId()` geht es um die in der `a2a_init()`- Methode definierte Projekt-ID.

Abschließend wird die ursprünglich definierte lokale Variable ausgegeben.

Die dazugehörigen Quellcodes sind in den folgenden *Code Listings* 5 und 6 zu finden.

```
bool a2a_wait4StartMain() {
	bool received = false;
	//wait until receive buffer is not empty
	if (!uart1_rxempty()) {
		//check the receive buffer
		input = uart1_getchar();
		if (input == A2A_STARTACTION)
			received = true;
		else
			return received;
	}
	return received;
}
```

**Code Listing 5: `a2a_wait4StartMain()` - Methode**

```
bool a2a_wait4ProjectId() {
	bool received = false;
	//wait until receive buffer is not empty
	if (!uart1_rxempty()) {
		//check the receive buffer
		input = uart1_getchar();
		if (input == projectId)
			received = true;
		else
			return received;
	}
	return received;
}
```

**Code Listing 6: `a2a_wait4ProjectId()` - Methode**

## Hilfsmethoden

Die `a2a_sendChar(char *data)`- Methode ist für das Senden eines einzigen Byte mit einem Symbol durch die *UART1*-Schnittstelle zuständig. Als Input bekommt diese Methode einen Pointer auf das gewünschte Zeichen.

Die `a2a_sendMessage(char []data)`- Methode sendet die gegebene Zeichenkette durch die *UART1*-Schnittstelle. Als Parameter bekommt die Methode ein Zeichen-Array. Dabei wird mittels einer `for`-Schleife in jeder Iteration die `a2a_sendChar()`- Methode für jedes Element des Zeichen-Arrays aufgerufen.

Die `a2a_strLength(char *s)`- Methode gibt die Elementanzahl des gegebenen Zeichen-Arrays zurück.

## Praktischer Einsatz der entwickelten Anwendung

Im Folgenden soll der praktische Einsatz der entwickelten *Nibo*-seitigen Anwendung am Beispiel eines Testprojekts `HelloAdapt2Android_LED` erklärt werden, das im Laufe der Implementierungsphase entstand. Dieses Testprojekt lässt die in dem *Nibo2*-Roboter eingebauten *LED*s im Kreis blinken.

Dabei wurde die in dem Unterkapitel *Einfügen eines neuen Nibo*-Projekts beschriebene Vorgehensweise verwendet. Der grundlegende Teil des Projekts ist die `main` –Methode, die die Hauptfunktionalität umfasst. Diese `main` –Methode ist in dem *Code Listing* 7 dargelegt. Hierbei wird angenommen, dass die notwendigen standardisierten *Nibo2*-Bibliotheken sowie der im *Code Listing* 1 gegebene Quellcode-Abschnitt in das vorliegende Projekt inkludiert wurde. Damit kann man sich jetzt nur auf den Methodeninhalt konzentrieren.

Als erstes werden alle Methoden zur Initialisierung aller für dieses Testprojekt relevanten *Nibo2*-Module aufgerufen, wie zum Beispiel `bot_init()`zur Initialisierung des Roboters, `spi_init()`zur Initialisierung der *SPI*-

Schnittstelle, `leds_init()` zur Initialisierung der *LED*s. Danach erfolgt die `a2a_initUART1()`- Methode zur Initialisierung des Bluetooth-Moduls und zum Einstellen derer Baudrate auf den benötigten Wert. Damit ist der Roboter zur Ausführung der eigentlichen Arbeit bereit. Ab diesem Moment nimmt die entwickelte Anwendung Einfluss auf das Projekt. Dabei wird eine `while`-Schleife verwendet, in welche die bekannte `a2a_wait4StartMain()`- Methode eingepackt wird. In jeder Iteration dieser Schleife wird durch die `_delay_ms()` -Methode der Wartemodus implementiert. Damit wird der Roboter warten bis die `a2a_wait4StartMain()`- Methode das erwartete Ergebnis liefert. Ist dies der Fall, verlässt die Anwendung die `while`-Schleife und greift auf die nachfolgende `a2a_init()`- Methode zu. Dabei werden die folgenden Parameter als Methoden-Input eingegeben:

- „*LED-Run*" als gewünschter Projektname,
- „7" als gewünschte Projekt-ID.

Nachdem der `a2a_init()`- Methodeninhalt ausgeführt wurde, soll die *hand shake* – Tätigkeit damit abgeschlossen werden. Ab diesem Moment wartet der Nibo2-Roboter auf die Signale zum Starten und Stoppen des initialisierten Projekts. Diese beiden Signale sind nur durch eine einzige Projekt-ID gekennzeichnet. Die Anwendung kehrt in einen weiteren Anwendungsbestandteil, wo die Ausführung des vorliegenden Projekts mit Hilfe der *Android*-App gesteuert wird. Dabei steht eine weitere aber unendliche `while`-Schleife zur Verfügung, die weitere Programmabschnitte beinhaltet. Hierbei handelt sich um eine weitere `while`-Schleife, die funktionsmäßig der Anfangs- `while`-Schleife ähnelt, aber mit dem Unterschied, dass in diesem Fall die `a2a_wait4ProjectId()`- Methode in dem Bedingungsblock verwendet wird. Dabei wartet der *Nibo2*-Roboter auf das Signal zum Projektstart. Ist das Startsignal angekommen, greift die Anwendung auf den letzten Quellcodeabschnitt mit einer weiteren `while`-Schleife zu und der Roboter führt das Projekt aus. Das Projekt wird ausgeführt, bis das Stoppsignal erhalten wird. In diesem Fall kehrt die Anwendung zum Anfang der bekannten unendlichen Schleife zurück.

```
int main() {
	sei();
	bot_init();
	spi_init();
	display_init();
	gfx_init();
	leds_init();
	//initialize uart1-port and set the baud rate
	a2a_initUART1();

	//wait until specific start char is received
	while (!a2a_wait4StartMain()) {
		_delay_ms(1);
	}
	//initialize Nibo-project
	a2a_init("LED-Run", '7');
	// Operation loop
	while (1) {
		//waits the project id is received
		while (!a2a_wait4ProjectId()) {
			_delay_ms(1);
		}
		while (!a2a_wait4ProjectId()) {
			//run project
			for (int i = 2; i < 8; i++) {
				leds_set_status(LEDS_GREEN, i);
				_delay_ms(50);
				leds_set_status(LEDS_OFF, i);
			}
		}
	}
}
```

**Code Listing 7: `main` -Methode des `Hello_Adapt2Android-LED` Beispielprojekts**

### 6.1.1 Entstandene Probleme

Für die Bearbeitung der vorliegenden Aufgabe konnte durch bereits vorhandene Kenntnisse im Bereich des *Nibo2*-Teilsystems eine gute Grundlage geschaffen werden. Hierbei war besonders die in dem Fach *C für eingebettete Systeme* gesammelte Erfahrung sehr hilfreich bei den Lösungsfindungen für die entstandenen Probleme. Hingegen erwies sich die Problemlösung im *Android*-Teilsystem als schwieriger, da hierfür eine intensive Einarbeitung in die Handhabung dieses Komplexes nötig war. Selbstverständlich kamen auch im Rahmen dieser Arbeit mit dem *Nibo2*-Teilsystem nicht triviale Probleme vor, die sich als besondere Herausforderungen der gesamten Arbeit darstellten.

Eines der während des Entwicklungsvorgangs entstandenen Grundprobleme war die falsche Konzeptionsauffassung des zu entwickelnden Teilsystems, was im Endeffekt einen Lösungsansatz lieferte, dessen Ergebnisse nicht mit den geplanten Ergebnissen übereinstimmten. Die Hauptursache dieses Problems war die bestandene Vorstellung, dass ein *Nibo2*-Roboter am Ende eine Möglichkeit gewährleisten soll, mehrere *Nibo*-Anwendungen zu haben und diese ausführen zu können. Der dabei zu entwickelnde Lösungsansatz sollte so angepasst werden, um diese *Nibo*-Anwendungen ansteuern zu können. Dies stellte sich in der Realität aufgrund der im Rahmen dieser Arbeit vorhandenen Ressourcen als nicht lösbar heraus. Wegen der Speicherplatzeinschränkung durch die Hardwarespezifikation war der *Nibo2*-Roboter nicht in der Lage, mehrere Anwendungen zu speichern. Ab diesem Moment wurde diese Tatsache als eine Konzeptionsbasis aufgefasst, was das grundlegende Herangehen an die Entwicklung änderte und in vielen Aspekten einen Einfluss auf den Endlösungsansatz hatte.

Das nächste Problem entstand im Laufe der Arbeit mit dem Kommunikationsprotokoll, dessen Beschreibung im Kapitel *Konzept* zu finden ist. Das Hauptziel bestand hierbei in der Entwicklung der Methode `a2a_init()`. In der Anfangsentwicklungsstufe enthielt diese Methode noch keinen Mechanismus zur automatischen Bestimmung des Endes von dem zu

generierenden Datenpaket durch die Ergänzung des Symbols „#" am Ende des jeweiligen Datenpakets. In Folge dessen erfolgten die ersten Testvorgänge während dieser Entwicklungsstufe ohne Verwendung dieses Mechanismus. Dabei wurde eine Seltsamkeit in der Funktion des *Nibo2*-Roboters entdeckt, die einen Anstoß gab, den gebrauchten Mechanismus schnell wie möglich zu implementieren. Diese Seltsamkeit war das Generieren von ungewöhnlichen und unerwarteten Symbolen durch den *Nibo2*-Roboter. Dieses Ereignis wurde dokumentiert und ist in den Abbildungen 24 und 25 zu sehen. Während der Beschäftigung mit diesem Ereignis wurde eine weitere Hilfsmethode `a2a_displayLogging()` zum so genannten „*Quasi-Logging*" erstellt, die den von der `a2a_init()`-Methode generierten Output auf dem Display des Roboters anzeigt. Damit konnte der ganze Generierungsablauf verfolgt werden und alle möglichen Übereinstimmungen des Methodenoutputs detektiert werden. Wie es aus den gegebenen Abbildungen zu erkennen ist, stimmen die ersten acht Zeichen (Bytes) mit dem geplanten Methodenoutput (= Projekt-ID + Projektname) überein, was aber bei dem Output-Rest nicht der Fall ist. Mit Hilfe von mehreren Versuchen, wurde aufgeklärt, dass die Art beziehungsweise Anzahl dieser unerwünschten Symbole direkt im Verhältnis mit dem Ein- sowie Ausschalten des Roboters waren. Also konnte das erwünschte Ergebnis mittels einigen Ein- und Ausschaltmanipulationen des Roboters geliefert werden, was in der Abbildung 26 zu sehen ist. Das hatte den direkten Einfluss auf die Arbeit an der *Android*-App, da diese manchmal eine lange Reihe von solchen Symbolen erhielt.

Leider wurde keine Ursache des oben beschriebenen Problems gefunden. Anderseits war die Ursachenaufklärung dieses Problems nicht Gegenstand dieser Arbeit.

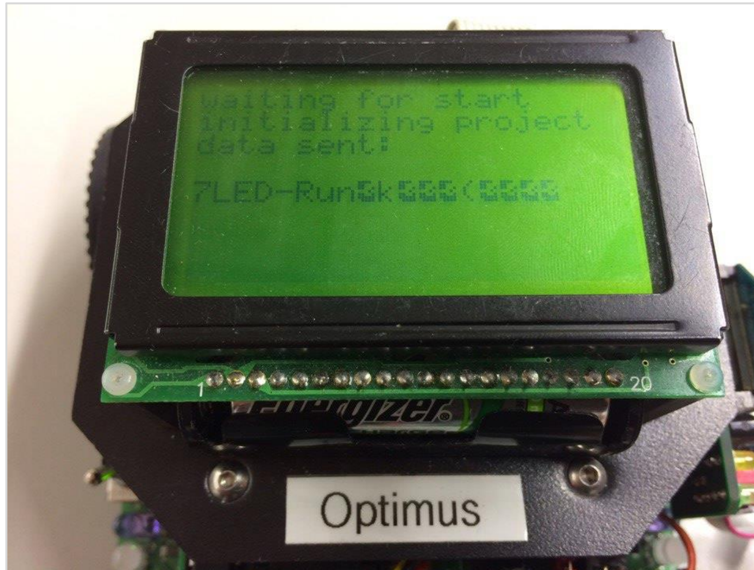



**Abbildung 24: Unerwarteter Output 1**

**Abbildung 25: Unerwarteter Output 2**

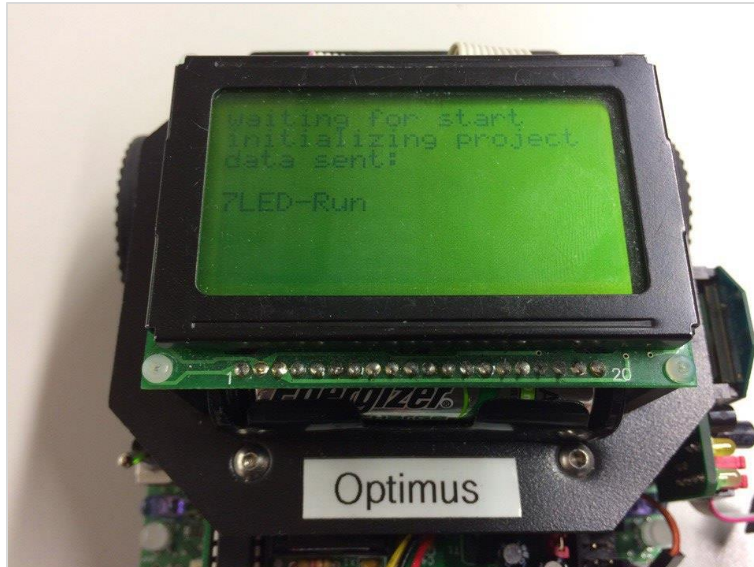



**Abbildung 26: Erwarteter Output**

Als Folge wurde der ganze Generierungsvorgang der richtigen Datenpakete nach der Realisierung des erwähnten Mechanismus erledigt. In der Abbildung 27 kann man das Endergebnis sehen.

**Abbildung 27: Output mit einem Endsymbol**

Das Endsymbol dient als ein Marker, der das Ende des jeweiligen Datenpakets kennzeichnet. Auf diese Weise wird nur das richtige Datenpaket an das *Android*-Teilsystem gesendet, unabhängig davon, ob die beschriebenen Ereignisse der Generierung von unerwünschten Symbolen aufgrund von unbekannten Ursachen stattfinden.

## 6.2 *Android*-Teilsystem

In diesem Unterkapitel wird ein Endlösungsansatz des *Android*-Teilsystems gegeben und auf die Hauptaspekte der Implementierungsphase eingegangen.

Anschließend werden die wesentlichen Klassen der Anwendung aufgelistet, ihre Klassendiagramme angegeben und näher beschrieben. Dabei ist zu beachten, dass für die von dem *Android SDK* standardisierten Klassen keine Klassendiagramme angegeben werden. Damit wird der Hauptfokus nur auf die selbst entwickelten Klassen gerichtet. Diese Übersicht wird bei der späteren Beschreibung der App hilfreich sein, weil auf diese Klassen mehrmals zurückgegriffen wird. Danach werden alle entwickelten Funktionsfenster aufgelistet und auf ihre Funktionalität sowie auf die dabei entstandene konzeptionelle Herangehensweise näher eingegangen. Abschließend werden die entstandenen Grundprobleme dargelegt und beschrieben.

### 6.2.1 Übersicht der wesentlichen Klassen

Im Folgenden wird eine Übersicht der selbst entwickelten Klassen dargelegt und beschrieben.

## `Nibo` und `NiboFactory`

Die Klasse `Nibo` wird als ein Kerndatenmodel im Rahmen des Systems betrachtet, das später im Laufe des gesamten Applikationsbetriebs als eine wichtige Funktionseinheit verwendet wird. Ein Objekt dieser Klasse vertritt einen einzigen *Nibo2*-Roboter, dessen Anwendung später im System an- bzw. abzumelden sowie anzusteuern ist. Dabei werden wichtigen Daten in dieser Klasse gespeichert, wie der Name dieses *Nibo2*-Roboters, die *Mac*-Adresse[1], der zu initialisierende Projektname sowie die Projekt-ID.

Die `NiboFactory` Klasse spielt eine besondere Rolle im ganzen System, da sie als ein Hauptspeicher verwendet wird, wo alle im System angemeldeten `Nibo`-Objekte zentral gespeichert werden und aus welchem auf diese Objekte von der Anwendung zugegriffen wird. Der eigentliche Speicherplatz ist dabei eine `ArrayList` aus den `Nibo`-Objekten. Ein Merkmal dieser Klasse ist jedoch von Bedeutung, dass diese Klasse unter der Verwendung des *„Singleton“ Design Patterns* entwickelt wurde.

Das *„Singleton“* stellt ein Entwurfsmuster dar, das die Erstellung nur eines einzigen Exemplars der vorliegenden Klasse garantiert und einen globalen Zugang zu diesem Exemplar bereitstellt. [10]

Die beschriebenen Klassen sind in der folgenden Abbildung 28 in einem *UML* Klassendiagramm grafisch dargestellt. Dabei ist zu sehen, welche Attribute und Methoden jede Klasse beinhaltet. Außerdem ist zu erkennen, welche Beziehung zwischen den beiden Klassen besteht. Dabei handelt es sich um die *Composition* (*Komposition)*, da die neu erstellten `Nibo`-Objekte in diesem Fall direkt im `NiboFactory`-Objekt gespeichert werden. Beim Löschen des `NiboFactory`-Objekts werden alle `Nibo`-Objekte auch gelöscht.

---

[1] In der Tat werden der Name und die *Mac*-Adresse des in diesem *Nibo2*-Roboter eingebauten Bluetooth-Moduls verwendet.

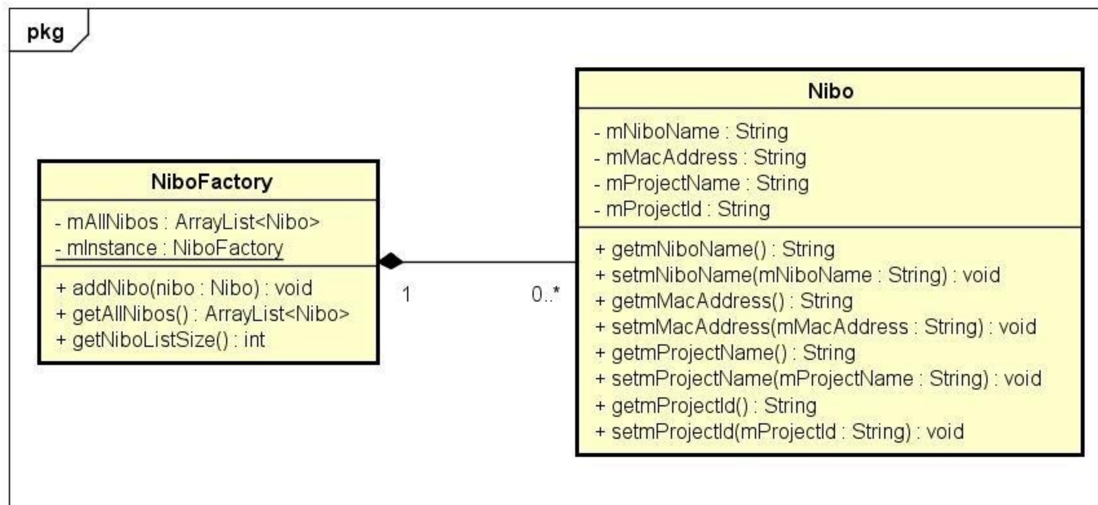



**Abbildung 28: `Nibo` und `NiboFactory` Klassen**

## `ConnectThread` und `ConnectedThread` (*DeviceListActivity*)

Ein grundlegender Baustein der ganzen *Android*-Applikation ist der Verbindungsaufbau per Bluetooth, durch welchen die Applikationshauptfunktionalität stattfindet. Dabei entstand der Lösungsansatz für die Bereitstellung der Bluetooth-Kommunikation mit den *Nibo2*-Robotern anhand des in der offiziellen Entwicklerdokumentation von *Android* dargelegten Beispielprinzips. [11]

Das Wesentliche ist dabei die Trennung des Kommunikationsvorgangs in zwei Stufen. Die erste Stufe umfasst den reinen Kommunikationsaufbau und die zweite führt die notwendige Datenverarbeitung nach dem erfolgreichen Kommunikationsaufbau durch. Dem jeweiligen Vorgang wird dabei ein *Thread* zugeordnet, um die Vorgänge logisch voneinander zu trennen und den Aufwand zu minimieren. Letztendlich entstanden die Klassen `ConnectThread` und `ConnectedThread`, die die beschriebenen Vorgänge ausführen. In der Abbildung 29 sind die beiden Klassen in einem Klassendiagramm grafisch dargestellt. Es ist zu sehen welche Attribute sowie Methoden diese Klassen beinhalten und aus welcher Super-Klasse sie erben.

Die beiden Klassen werden später in der *DeviceListActivity* während des Betriebsablaufs verwendet, um die *Nibo*-Projekte in dem System anzumelden.

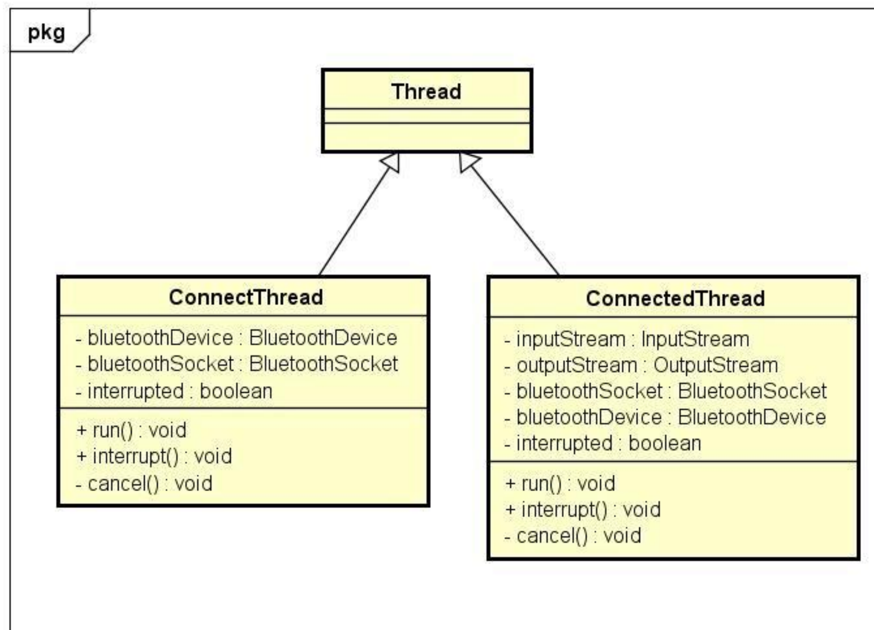



**Abbildung 29: `ConnectThread` und `ConnectedThread` (*DeviceListActivity*)**

## `ConnectThread` und `ConnectedThread` (*ProjectListActivity*)

Die weiteren zwei Klassen realisieren ebenfalls das im früheren Absatz beschriebene Trennungsprinzip des Kommunikationsaufbaus von der Datenverarbeitung. Darüber hinaus unterscheiden sich diese Klassen aber von den vorherigen beiden Klassen durch die Verwendung in einer anderen *Activity* (*ProjectListActivity*) sowie durch die Verwendung für andere Zwecke. Die vorliegenden `ConnectThread` und `ConnectedThread` Klassen beschäftigen sich mit dem Starten und Stoppen des initialisierten *Nibo*-Projekts. Dabei wird die Projekt-ID mittels der `ConnectedThread` Klasse an den *Nibo2*-Roboter gesendet. In der folgenden Abbildung 30 sind die beiden Klassen in einem Klassendiagramm grafisch dargestellt.

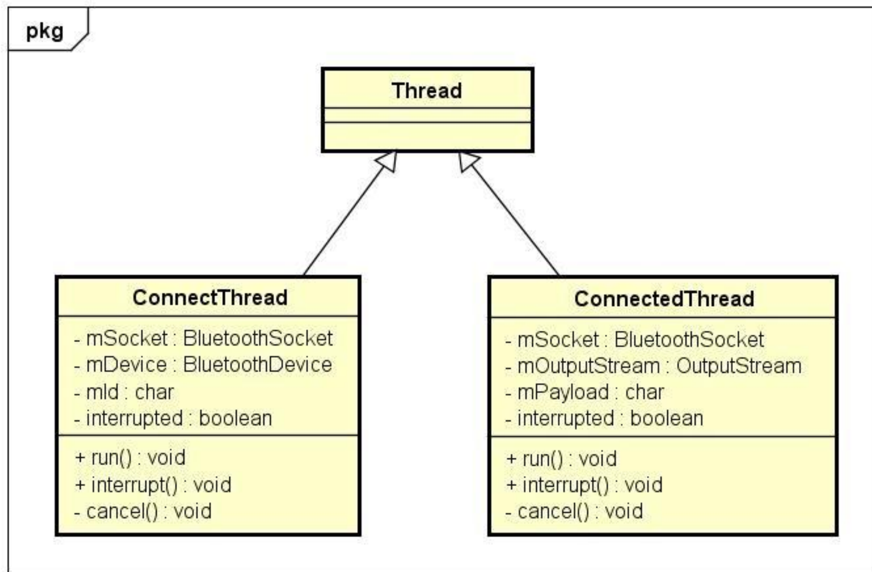



**Abbildung 30: `ConnectThread` und `ConnectedThread` (*ProjectListActivity*)**

### 6.2.2 Die *Activities*

Im Folgenden wird eine Übersicht der entwickelten *Activities* dargelegt und beschrieben.

#### *MainActivity*

Die *MainActivity* stellt einen Kernbestandteil der ganzen *Android*-App dar. Dabei wird die Systemfunktionalität mittels dieser *Activity* zur Verfügung gestellt. Das Layout wurde im Laufe der Implementierungsphase ständig geändert und die Endvariante dieses Layouts kann man in der folgenden Abbildung 31 sehen.

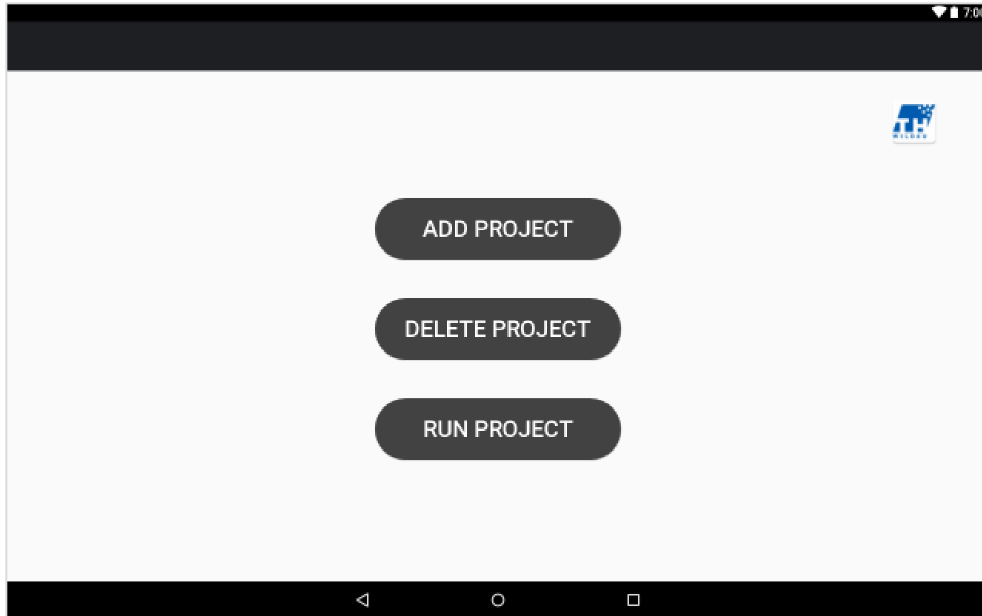



**Abbildung 31:** ***MainActivity*** **(Screenshot)**

Wie es im Kapitel *Storyboard* dargelegt wurde, besteht diese *Activity* grundsätzlich aus drei aktiven *View Elements*, die *Buttons.* Beim Betätigen des jeweiligen *Buttons* werden die folgenden Aktivitäten ausgeführt:

- bei Betätigen des *ADD PROJECT-Buttons* wird ein Scanvorgang zur Findung der Bluetooth-Geräte durchgeführt und im erfolgreichen Fall in die *DeviceListActivity* weitergeleitet;
- bei Betätigen des *DELETE PROJECT-Buttons* wird in die *DeleteProjectActivity* weitergeleitet;
- bei Betätigen des *RUN PROJECT-Buttons* wird in die *ProjectListActivity* weitergeleitet.

Ein wichtiger Aspekt dabei ist, dass beim Laden der *MainActivity* Klasse durch die Methode `checkNiboListState()` geprüft wird, ob schon initialisierte *Nibo*-Projekte vorhanden sind. Dafür wird die bekannte *ArrayList* in der *NiboFactory* nachgeprüft. Ist dies der Fall, werden die *Buttons DELETE PROJECT* und *RUN PROJECT* in dunkel-grauer Farbe gefärbt (Abbildung 31). Andernfalls werden sie in hell-grauer Farbe gefärbt und inaktiv sein. Beim ersten Starten

der *Android*-App sind diese beiden *Buttons* ursprünglich hell-grau und inaktiv, da noch keine *Nibo*-Projekte vorhanden sind (Abbildung 32).

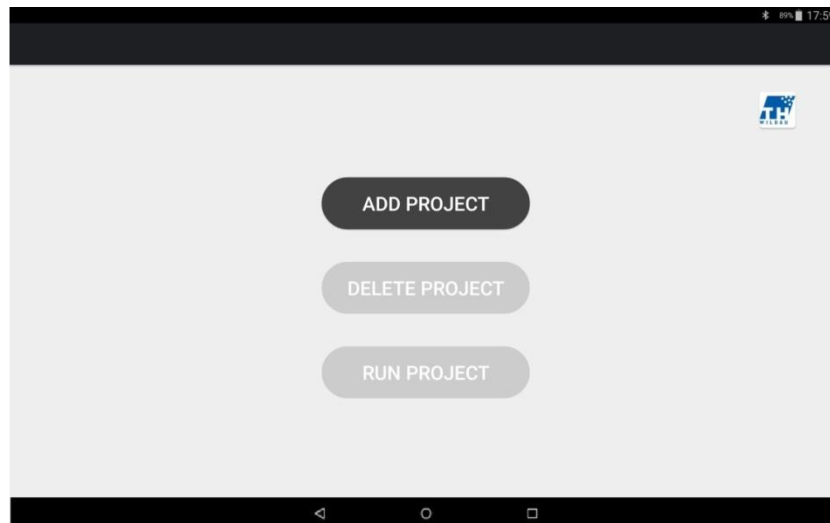



**Abbildung 32: *MainActivity* (mit ausgeblendeten Knöpfe)**

Der nächste wichtige Aspekt dieser *Activity* ist der bekannte Scanvorgang, der das Bluetooth-Medium scannt und nach den Bluetooth-Geräten sucht. Der Scanvorgang wurde mit Hilfe von den `BroadcastReceiver` und `IntentFilter` Klassen realisiert.

Die *Broadcasts* im Kontext der *Android*-App werden als ein Nachrichtensystem verwendet. Dabei können diese *Broadcasts* entstehen, wenn ein Systemereignis stattgefunden hat, zum Beispiel wenn der Bluetooth-Adapter des *Android*-basierten Geräts ein- bzw. ausgeschaltet wird oder wenn das *Android*-System startet. Dabei dient der `BroadcastReceiver` als ein Überwacher von solchen Systemevents. Mittels der `IntentFilter` Klasse können die Systemereignisse einem bestimmten `BroadcastReceiver` Objekt zugewiesen werden, die von diesem `BroadcastReceiver` Objekt zu überwachen sind. Darüber hinaus kann das weitere Applikationsverhalten in den Fällen, wenn beliebige Systemevents vorgekommen sind, geregelt werden. Außerdem ist es möglich die benutzerdefinierten Ereignisse festzustellen und diese dann später zu verwenden. Im Rahmen dieser Arbeit wurden aber nur die

vorhandenen Systemevents verwendet, die in dem *Bluetooth API* angegeben sind. [12]

Die folgenden Systemevents wurden in der *MainActivity* festgestellt und eingesetzt:

- Systemevent `ACTION_STATE_CHANGED` der `BluetoothAdapter` Klasse, das den Status des lokalen Bluetooth-Adapters überwacht,
- Systemevent `ACTION_DISCOVERY_STARTED` der `BluetoothAdapter` Klasse, das das Starten des Scanvorgangs überwacht,
- Systemevent `ACTION_DISCOVERY_FINISHED` der `BluetoothAdapter` Klasse, das das Stoppen des Scanvorgangs überwacht,
- Systemevent `ACTION_FOUND` der `BluetoothDevice` Klasse, das die Findung eines Bluetooth-Gerätes bezeichnet.

### *DeviceListActivity*

Die vorliegende *DeviceListActivity* sowie die *ProjectListActivity* und *DeleteProjectActivity* verfügen über ein gemeinsames aufbaustrukturelles Merkmal. Jede *Activity* besitzt das häufig verwendete *View Element* nämlich *ListView* zum Inhalt, das eine scrollbare Liste aus der vorhandenen Objektmenge zur Verfügung stellt. In der folgenden Abbildung 33 ist das Layout (eine *XML*-Datei) des *DeviceListActivity* dargelegt, wobei das *ListView* in der Mitte als ein viereckiger Container zu erkennen ist. Ursprünglich beinhaltet das *ListView* keine Elemente. Das Inhaltsauffüllen der Liste erfolgt durch eine Strukturaufbaubesonderheit. Die Besonderheit dieses *View Elements* besteht darin, dass jedes Listenelement durch eine Adapterklasse in die Liste hinzugefügt wird. Dabei werden zuerst die einzelnen benötigten *View Elemente* zu einem End-*ViewElement* konvertiert, das später hinzuzufügen ist.

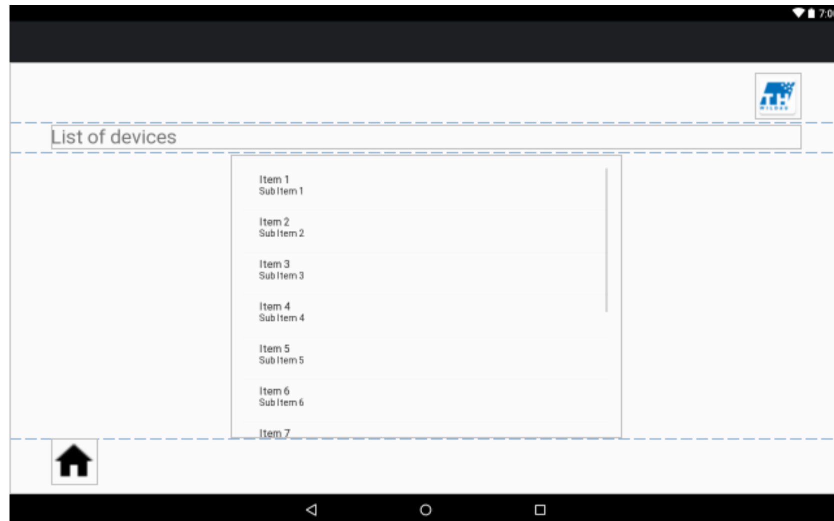



**Abbildung 33:** `activity_device_list.xml`

Das Layout des End-*ViewElements* soll dabei in einer separaten *XML*-Datei definiert werden. Das Konvertieren sowie das Einfügen des einzelnen Elements in die Liste übernimmt dabei die `getView()`-Methode der `DeviceListAdapter`-Klasse. [13]

Zum Beispiel kann man das Element für das vorliegende *ListView* Objekt der *DeviceListActivity* in der folgenden Abbildung 34 sehen. Dabei sind zwei *TextView* für den Namen und die *Mac*-Adresse des gefundenen *Nibo*-Roboters und ein *Button* zum Starten der Initialisierungstätigkeit zusammengefasst.

**Abbildung 34:** `list_item_project_add.xml`

Das endgültige Ergebnis ist in der folgenden Abbildung 35 zu sehen.

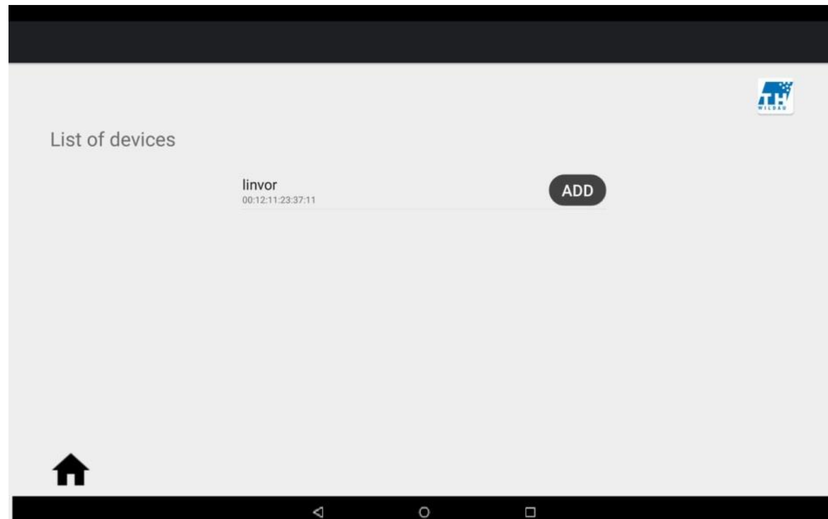



**Abbildung 35: *DeviceListActivity* (Screenshot)**

Ein wesentlicher Aspekt dieser *Activity* ist das Starten des wichtigen Systemfunktionsvorgangs nach dem Betätigen des vorliegenden *ADD-Buttons*. Dabei handelt es sich um den bekannten Initialisierungsvorgang. An dieser Stelle ist wichtig zu erwähnen, dass im Rahmen dieses Projekts die von *Android* angebotene standardisierte *Bluetooth API* verwendet wurde, da alle in den *Nibo2*-Robotern eingebauten *Blue Module* nur durch diese Bibliothek programmierbar sind. [14]

Nach dem Betätigen des *Buttons* wird die `manageConnectThread()` –Methode aufgerufen, die ein Objekt der bekannten `ConnectThread` –Klasse erstellt und dieses danach startet. Als Parameter bekommt die Methode das Objekt der `BluetoothDevice`-Klasse, das zum ausgewählten *Nibo2*-Roboter zugeordnet ist und zur `ConnectThread` –Klasse übergeben wird. Dabei wird diese Methode als `synchronized` deklariert. Damit wird eine Fallberücksichtigung vorgenommen, dass der Methodenablauf „*threadsicher*“ ist. Der Begriff „*threadsicher*“ bedeutet, dass die Methodenaufrufe in den neuen *Threads* immer sequentiell nacheinander ausgeführt werden. Das `ConnectThread` –Objekt beschäftigt sich mit dem Kommunikationsaufbau mit dem ausgewählten *Nibo2*-Roboter. Die Hauptsache ist dabei die Erstellung eines *`BluetoothSockets`* zum ausgewählten Roboter, das zum reinen

Datenaustausch benötigt wird. Dies erfolgt durch den *Code*-Abschnitt, der in dem folgenden *Code Listing* 8 zu sehen ist.

```
tmp =
(BluetoothSocket) bluetoothDevice.getClass()
            .getMethod("createRfcommSocket",
            new Class[]{int.class}).invoke(bluetoothDevice, 1);
```

**Code Listing 8: Erstellung eines `BluetoothSockets`**

Nachdem das benötigte *Socket* erfolgreich erstellt wurde, führt die Anwendung die Verbindungsversuche zu diesem *Socket* aus. Falls die Verbindung erfolgreich war, wird die `manageConnectedThread`-Methode aufgerufen, die das Objekt der `ConnectedThread`-Klasse erstellt und startet. Die `manageConnectedThread`-Methode wird jedoch als `synchronized` deklariert. Die Codeausführung des `ConnectThread`–Objekts wird mit dem `interrupt()`-Methodenaufruf abgebrochen, da die Verbindung als erfolgreich gekennzeichnet wurde.

Ab diesem Moment übernimmt das `ConnectedThread-`Objekt die weiteren Aktivitäten zum Datenaustausch. Als erstes werden die Objekte der `InputStream`-Klasse für die Empfangsdaten sowie der `OutputStream`-Klasse für die Sendedaten zum erstellten *Socket* zugeordnet. Durch das `OutputStream-`Objekt wird das bekannte vordefinierte Startsignal zum *Nibo2*-Roboter gesendet. Innerhalb der `ConnectedThread`-Klasse erledigt die nachfolgende Aktion zum Datenempfang bzw. zur Datenverarbeitung ein extra *Thread* –Objekt. Dieses *Thread* –Objekt liest die Daten aus dem `InputStream-`Objekt, die von dem *Nibo2*-Roboter gesendet wurden. Nach dem erfolgreichen Datenempfang wird an dieser Stelle das neue Objekt der `Nibo`-Klasse erstellt und dabei die erhaltenen Daten zu diesem Objekt zugeordnet. Überdies wird das erstellte `Nibo`-Objekt in der *ArrayList* `mAllNibos` des `NiboFactory`–Objekts gespeichert. Abschließend werden die verwendeten Objekte terminiert.

Damit soll die Initialisierungsphase eines *Nibo*-Projekts abgeschlossen werden. Das vorliegende Projekt wird ab diesem Moment aufrufbar sein.

Für den Fall, wenn ein neuer Suchvorgang nach dem Betätigen der *ADD PROJECT – Buttons* in der *MainActivity* von dem Anwender initiiert wird, wird der *Nibo2*-Roboter mit dem schon angemeldeten Projekt auf dem *ListView* des *DeviceListActivity* nicht eingeblendet. Dafür ist die `checkStateOfStorage()`-Methode zuständig.

### *ProjectListActivity*

Das endgültige Layout der *ProjectListActivity* ist in der Abbildung 36 zu sehen. Dabei wird die Vorgehensweise der wesentlichen Systemfunktionstransaktionen jedoch von den lokalen `ConnectThread-` sowie `ConnectedThread`-Klassen übernommen. Im Unterschied zu den beiden Klassen aus der *DeviceListActivity*, werden durch das `OutputStream`-Objekt die Daten zum *Nibo2*-Roboter nur gesendet. Damit werden das Starten sowie das Stoppen des vorhandenen *Nibo*-Projekts gewährleistet.

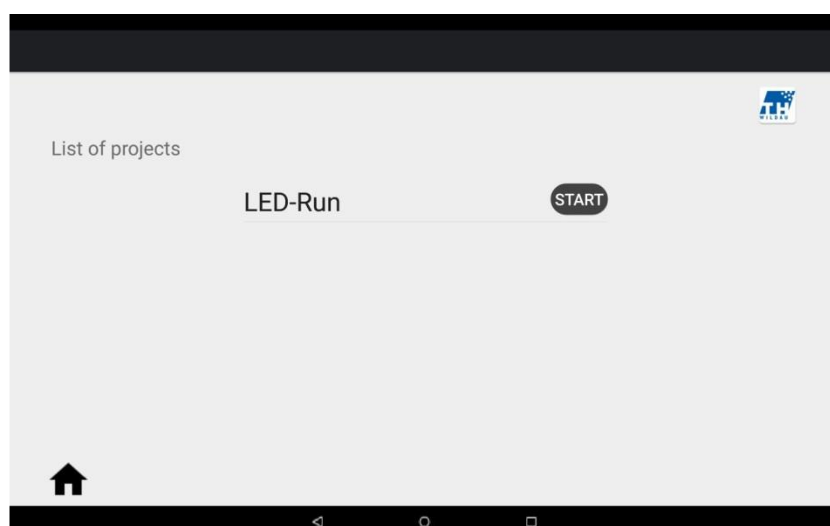



**Abbildung 36: *ProjectListActivity* (Screenshot)**

### *DeleteProjectActivity*

Das endgültige Layout der *DeleteProjectActivity* ist in der Abbildung 37 zu erkennen. Beim Betätigen des *DELETE-Buttons*, wird das ausgewählte Projekt aus dem System gelöscht und wird nicht mehr aufrufbar. Dabei wird das *ListView* dynamisch aktualisiert. Um wieder das gelöschte Projekt aufrufen zu können, soll dieses dann neu angemeldet werden und davor der bestimmte *Nibo2*-Roboter neugestartet werden.

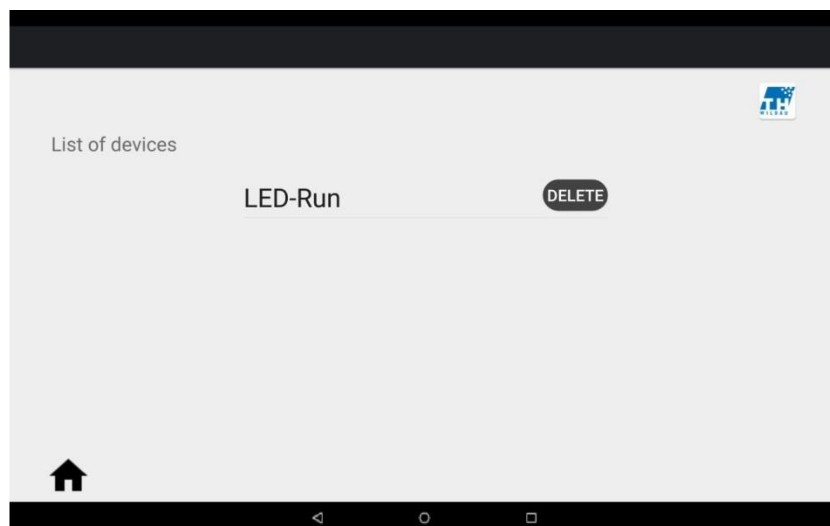



**Abbildung 37: *DeleteProjectActivity* (Screenshot)**

### 6.2.3 Entstandene Probleme

Das grundlegende im Laufe der Implementierungsphase entstandene Problem war der Umgang mit dem mehrfachen Kommunikationsaufbau per Bluetooth durch ein `BluetoothSocket` zu einem festdefinierten *Nibo2*-Roboter. Dieses Problem trat am Anfang der Implementierungsphase der zu entwickelnden *Android*-App auf. Hierbei wurde folgender Ablauf der ganzen Transaktion zum

Kommunikationsaufbau, zum Datenaustausch sowie zur Datenverarbeitung definiert:

1. Erstellung eines neuen `BluetoothSocket` für den vorliegenden *Nibo2*-Roboter,
2. Aufbau einer Verbindung zu diesem Socket durch die `BluetoothSocket.connect()`-Methode,
3. Datenverarbeitung der Empfangsdaten.

Im anschließenden Fall des erneuten Kommunikationsaufbaus durch das vorliegende Socket kam die folgende *Exception* vor:

```
"java.io.IOException: read failed, socket might closed, read
ret: -1".
```

Es wurde versucht dieses entstandene Problem folgendermaßen zu lösen: das verwendete *Socket*-Objekt wurde nach allen Datenverarbeitungsvorgängen terminiert und damit vor dem weiteren Kommunikationsaufbau ein neues *Socket* zum bestimmten Roboter erstellt. Dies brachte leider wenig Erfolg. Dabei trat diese *Exception* trotzdem ein und außerdem musste der *Nibo2*-Roboter neugestartet werden. Nach weiterer Recherche konnte die Ursache für das vorliegende Problem gefunden werden. Dank eines in der *Stack Overflow Community* gefundenen Artikels konnten die Problemursachen verstanden werden und damit der gegebene Lösungsvorschlag in der zu entwickelnden *Android*-App erfolgreich eingesetzt werden. [15] In diesem Artikel wurde behauptet, dass die *Android*-basierten Geräte mit der Betriebssystemversion höher als 4.2 anfällig für die bekannte *Exception* sind. Das Wesentliche aus diesem Artikel ist dabei, dass die aus der *Bluetooth API* standardisierte `createRfcommSocketToServiceRecord()`- Methode zur Erstellung des `BluetoothSockets` den größten Einfluss auf die gesamte Funktionalität hatte. In dem folgenden *Code Listing* 9 ist die ursprüngliche Vorgehensweise zur Erstellung des `BluetoothSockets` zu sehen, auf welche letztendlich entschieden wurde zu verzichten.

```
tmp = bluetoothDevice.createRfcommSocketToServiceRecord(uuid);
```

**Code Listing 9: Erstellung des `BluetoothSockets` (alte Variante)**

Bei der Erstellung eines neuen Sockets wird seinem Paramater `mPort` standardmäßig der Wert von „-1“ zugewiesen, worin die Hauptursache für das Problem lag. Die Lösung ist dabei das Umsetzen des Parameterwerts auf andere Werte, zum Beispiel „1“. Jedoch verfügt die `createRfcommSocketToServiceRecord()`- Methode nur über einen einzigen Input-Parameter, nämlich das Objekt der `UUID`-Klasse und besitzt keinen Parameter für den `mPort`. Daher sollte ein extra Lösungsweg gefunden werden, um mit dem `mPort` Parameter umgehen zu können. Als Lösung wurde in dem Artikel festgestellt, dass die standardisierte *Bluetooth API* eine Methode namens `createRfcommSocket()` beinhaltet, die den gebrauchten Parameter `mPort` als Input enthält. Die Besonderheit ist dabei, dass diese Methode *versteckt* (*hidden*) ist. *Versteckt* heißt, dass es unmöglich ist, diese Methode direkt aufrufen zu können, da keine Methodeninstanz existiert. Um auf diese Methode zugreifen zu können, wurde sich an den so genannten *reflection* Vorgang gewendet, dessen Ablauf hauptsächlich die Methode `java.lang.reflect.Method.invoke()` übernimmt. Dabei sind beim *reflection* Vorgang die notwendigen Aktivitäten gekennzeichnet, die den Methodenaufruf in diesem Ausnahmefall gewährleisten. Hierzu wird, wie in dem *Code Listing* 8 dargelegt wurde, zuerst ein Objekt der Klasse `Class` durch die `getClass()`-Methode der Klasse `Object` ermittelt. Die Instanz der Klasse `Class` stellt eine Repräsentierung der beliebigen Klasse in einer *Java* Applikation dar. [16] In diesem Fall handelt es sich um die `BluetoothDevice`-Klasse. Um auf die vorhandenen Methoden dieser ermittelten Klasse indirekt zugreifen zu können, wird die vorhandene Methode `getMethod()` der Klasse `Class` verwendet. Dabei werden zwei Parameter benötigt. Beim ersten Parameter wird der Name der gewünschten Methode eingegeben, der in diesem Fall „`createRfcommSocket`“ ist. Der zweite Parameter repräsentiert den

Datentyp dieser Methode für den bekannten Parameter `mPort`, welcher in diesem Fall vom Datentyp `Integer` ist. Abschließend wird durch die erwähnte Methode `invoke()` die aufzurufende Methode ausgeführt. Damit wurde dieses Problem bei dem Kommunikationsaufbau behoben. [17]

Ein weiteres entstandenes Problem war mit der Effizienz des Funktionsaufwands verbunden. Hierbei ergab sich das vorliegende Problem durch das Auftreten der folgenden *Exception*:

```
I/Choreographer(xxxx): Skipped xxx frames!  The application may
be doing too much work on its main thread.
```

Der Grund für diese *Exception* war, wie der *Exception*-Inhalt lautet, dass das aktuelle `Thread`-Objekt zu viel Arbeit gleichzeitig erledigt. Das passiert zum Beilspiel, wenn die umfangreichen mathematischen Berechnungen in der Applikation stattfinden. Im Rahmen der zu entwickelnden App trat diese *Exception* hauptsächlich innerhalb des Funktionsablaufs von dem `ConnectedThread`-Objekt während der Initialisierungsphase ein. Dabei lag das eigentliche Problem bei dem Datenlesen aus dem `InputStream`-Objekt. Ursprünglich wurde kein extra `Thread`-Objekt beim Datenlesevorgang verwendet. Aber nach dem mehrmaligen Auftreten der vorliegenden *Exception* und nach dem Aufklären aus einem Artikel aus der *Stack Overflow Community*, wurde entschieden ein extra `Thread`-Objekt einzusetzen, was sich letztendlich auf den Endlösungsansatz auswirkte. [18]

Ein weiteres Problem ergab sich mit dem Umgang bei den dynamischen Zustandsänderungen von den *View* Elementen. Dabei trat in Folge dessen immer ein Absturz der gesamten App ein. Das Wesentliche ist dabei, dass alle dynamischen Änderungen der *View* Elemente nur in dem Haupt-`Thread` durchgeführt werden sollen, da ansonsten ein Funktionsabsturz der gesamten App eintritt. Unter dem Haupt-`Thread` ist dabei das `UI-Thread` der Anwendung bezeichnet, an das mittels der `runOnUiThread()`-Methode zugegriffen wird.

Der letzte Punkt ist nicht als ein entstandenes Problem zu betrachten, sondern als ein wichtiger Schritt zur Anwendungsoptimierung. Dieser Optimierungsschritt hatte im Endeffekt auch einen wichtigen Einfluss auf den Endlösungsansatz.

Dabei handelt es sich um das Herangehen zur Datenspeicherung in der App. Ursprünglich wurden anstatt des *„Singleton" Design Patterns* die *GSON*-Bibliothek zum Umgang mit den *JSON*-Objekten verwendet. [19] Hierzu wurden die benötigten Daten (die zurzeit als Attribute der `Nibo`-Klasse definiert sind) in dem *JSON*-Objekt zusammengefasst. Dieses *JSON*-Objekt wurde danach zu einem *String* konvertiert und letztendlich in einer gemeinsamen *String-ArrayList* gespeichert. Abschließend wurde diese *ArrayList* zur Datenübermittlung zwischen den *Activities* verwendet. So sah zum Beispiel das generierte *JSON*-Objekt für einen *Nibo2*-Roboter aus, der im Laufe des ganzen Betriebs verwendet wurde:

```
{"nibo": {
  "name": "linvor",
  "mac-address": "00:12:11:23:37:11",
  "function-name": "LED-Run",
  "function-id": "7"
}}
```

**Abbildung 38: *JSON*-Objekt**

Der Grund für die Verwendung der *JSON*-Objekte zur Datenspeicherung ergab sich aus einer Tatsache, die eine Spezifik der Entwicklung unter der *Android*-Plattform bezeichnet. Dabei handelt es sich um die Datenübermittlung zwischen den *Activities*. Ein standardisierter Weg ist dabei die Verwendung der Klasse `Intent`, die normalerweise zur Umleitung von einer *Activity* zu einer anderen eingesetzt wird. So wurde zum Beispiel das Umleiten in die *ProjectListActivity* nach dem Betätigen des *RUN PROJECT Buttons* durch die `Intent`-Klasse in der *MainActivity* realisiert:

```
mRunProjectButton.setOnClickListener(new View.OnClickListener(){

     @Override
     public void onClick(View v) {

            Intent   newIntent   =   new   Intent(MainActivity.this,
            ProjectListActivity.class);

            startActivity(newIntent);

     }

});
```

**Code Listing 10: Umleiten in andere *Activity***

Dabei erfolgt die Datenübermittlung durch die zusätzliche Anheftung der gewünschten Daten in dem `Intent`-Objekt. Dies erfolgt mittels der Methode `putStringArrayListExtra()`, um die vorliegende *ArrayList* mit den JSON-Objekten zu übermitteln. Genau das war der Grund die *JSON*-Objekte überhaupt zum *String* Datentyp zu konvertieren. In dem folgenden *Code Listing* 11 ist zu sehen, wie beispielsweise die `mProjectList` in die `ProjectListActivity` übermittelt wird:

```
mRunProjectButton.setOnClickListener(new View.OnClickListener(){

     @Override
     public void onClick(View v) {

            Intent   newIntent   =   new   Intent(MainActivity.this,
            ProjectListActivity.class);

            newIntent.putStringArrayListExtra("project.list",
            mProjectList);

            startActivity(newIntent);

     }

});
```

**Code Listing 11: Umleiten in andere *Activity* mit der Datenanheftung**

Durch die Anwendung der *GSON*-Bibliothek entstand ein merklich gesteigerter Funktionsaufwand bei dem zweiseitigen *Parsing* der *JSON*-Objekte, wodurch die Komplexität des Quellcodes zunahm und an Anschaulichkeit verlor.

Der Einsatz des „*Singleton*“ *Design Patterns* hatte einen positiven Einfluss auf den gesamten Funktionsaufwand sowie auf die Quellcode-Anschaulichkeit beziehungsweise –Komplexität. Ab diesem Moment wurde auf die weitere Verwendung der *GSON*-Bibliothek verzichtet.

# 7 Fazit und Ausblick

Im folgenden Kapitel wird die gesamte Arbeit in einem Fazit zusammengefasst und abschließend wird auf die weiteren Entwicklungsmöglichkeiten der fertigen Lösung eingegangen.

## 7.1 Fazit

Während der gesamten Umsetzungsphase wurde festgestellt, dass einige vordefinierte funktionale Anforderungen an das zu entwickelnde System nicht erfüllt oder teilweise erfüllt wurden.

Als erstes kommt die funktionale Anforderung **F7- „Rückmeldung“.** Dabei handelt es sich hauptsächlich darum, dass die *Nibo2*-Anwendung sich beim Erledigen ihres Betriebs (in diesem Fall beim Erledigen des *Nibo*-Projekts) eine Bestätigungsnachricht ihres Betriebsendes zur *Android*-Anwendung sendet. Es war technisch unmöglich diese Anforderung aus folgendem Grund zu realisieren: der theoretische Lösungsansatz mit dem einschließenden Einsatz eines extra `Thread`-Objekts für das Empfangen der Bestätigungsnachrichten von jedem *Nibo2*-Roboter ist technisch nicht möglich, da die parallelen Bluetooth-Verbindungen nicht von der verwendeten *Bluetooth API* unterstützt werden. Des Weiteren wurde der Endlösungsansatz unter der Berücksichtigung der vorliegenden Tatsache so angepasst, damit das wesentliche Projektvorhaben funktionsfähig ist. Darüber hinaus wurde auf die Realisierung dieser Anforderung verzichtet, da diese hauptsächlich keinen großen negativen Einfluss auf das gesamte System hatte.

Die weitere funktionale Anforderung, die aber teilweise erfüllt wurde, ist die Anforderung **F5- „Broadcast“**. Die ursprüngliche Vorstellung der parallelen

Kommunikation zwischen der *Android*-App und mehreren *Nibo2*-Robotern sah zu Beginn anders aus und änderte sich während der Umsetzungsphase, um das wesentliche Projektvorhaben zu realisieren. Diese ursprüngliche Vorstellung stellte dar, dass durch das Betätigen des einzelnen *Buttons* mehrere *Nibo2*-Anwendungen **gleichzeitig** angesteuert werden könnten. Diese Anforderung wurde in dem vorliegenden Endlösungsansatz nicht auf diese Weise umgesetzt, was aber keinen Einfluss auf die fertige Lösung hatte. Der Endlösungsansatz ermöglicht die gezielte **sequenzielle** Kommunikation mit mehreren *Nibo2*-Robotern. Dabei änderte sich die ursprüngliche Vorstellung des Projektvorhabens aufgrund der entdeckten Plattformspezifikation, wie in der vorangegangenen Ausführung beschrieben wurde.

Resümierend stellt die fertige Anwendung eine Schnittstelle dar, deren Hauptfunktionalität die Ansteuerung der Robotersystemanwendungen des *Nibo2*-Roboterbausatzes gewährleistet. Dabei ist die Hauptanwendung in zwei Teilanwendungen logisch getrennt. Die *Android*-seitige Teilanwendung ist in der Lage per Bluetooth mit mehreren *Nibo2*-Robotern eine Kommunikation zu gewährleisten und die *Nibo*-seitige Teilanwendung wiederum ist auch in der Lage mit den *Android*-basierten Geräten zu kommunizieren. Dabei können die existierenden *Nibo*-Projekte an die Hauptanwendung angebunden werden, sowie dynamisch in der *Android*-Teilanwendung an- und aus dieser Teilanwendung abgemeldet werden. Die *Android*-seitige Teilanwendung ermöglicht das dynamische Starten beziehungsweise das Stoppen der angemeldeten *Nibo*-Projekte. Die *Nibo*-Teilanwendung ist ihrerseits in der Lage sich in einem Wartemodus zu befinden bis der Betriebsstart von der *Android*-Teilanwendung initiiert wird.

Der in der fertigen Lösung angebotene Lösungsansatz zum Bluetooth-Kommunikationsaufbau mit dem *Nibo2*-Roboterbausatz kann als Grundlage für die Weiterentwicklungen im Gebiet des Datenaustausches per Bluetooth verwendet werden. Dabei sind nicht nur die beleuchteten Aspekte beim Bluetooth-Kommunikationsaufbau von Bedeutung, die im Kapitel *Umsetzung* beschrieben wurden, sondern auch das dabei verwendete Prinzip.

Im Laufe der Arbeit wurden die bestandenen Kenntnisse im Umgang mit der Programmiersprache *C* erweitert beziehungsweise gesichert, die ursprünglich aus dem Fach *C für eingebettete Systeme* stammen. Mittels der Lösungsfindung für die entstandenen nicht trivialen Aufgaben, halfen die dabei erworbenen Erfahrungen sich in die Spezifik dieser Programmiersprache zu vertiefen.

Die Entwicklung der mobilen Applikationen unter dem *Android* Betriebssystem stellt heutzutage eine der stark wachsenden Branchen in unserer modernen Welt dar. Damit öffneten die im Rahmen dieser Arbeit erworbenen Erfahrungen die Türen in der Welt der Entwicklung der mobilen Applikationen und darüber hinaus machten sie das mögliche Potential dieser Branche sichtbar. Diese Arbeit diente als ein vortrefflicher Motivator und weckte das Interesse für die Fortführung der Selbstvervollkommnung in dieser Branche. Diese Erfahrungen wurden hauptsächlich durch die eigentliche Lösungsfindung für die im Laufe der Umsetzungsphase erstandenen Probleme erworben.

Außerdem ist zu erwähnen, dass eine wichtige grobe Vorstellung erhalten wurde, die mit einem heutzutage in unserer modernen Welt gefragten Aspekt verbunden ist. Es handelt sich um einen technologischen Vorgang zur Vereinigung der verschiedenen technischen Systeme in ein gemeinsames System, das im Endeffekt eine mächtige Synergie im ausgewählten Einsatzgebiet gewährleisten kann. Durch die vorliegende Arbeit konnten wichtige Aspekte beleuchtet werden, die während der Realisierung derartiger Projekte entstehen können. Im Rahmen dieser Arbeit ist das zum Beispiel die Konzipierung und Implementierung des Kommunikationsprotokolls zum Datenaustausch zwischen den beiden Teilsystemen der entwickelten Hauptanwendung.

## 7.2 Ausblick

Eine mögliche Weiterentwicklung wäre der Einsatz von dem so genannten *„clean architecture“* Prinzip. Dabei ist gezielt das *MVP (Model-View-Presenter) Pattern* gemeint. [20]

Der Kernpunkt ist dabei die logische Trennung des *Presentation Layer* (nämlich der *GUI*) von der Hauptlogik der App. Damit wird die gesamte zu entwickelnde App in drei Hauptteile gegliedert:

- *Model* erledigt die Hauptlogik der App,
- *View* ist nur für die grafische Darstellungen auf der *GUI* zuständig und wird von der Hauptlogik getrennt,
- *Presenter* ist als eine Brücke zwischen *Model* und *View* zu verstehen, die Interaktionen zwischen beiden Niveaus ansteuert.

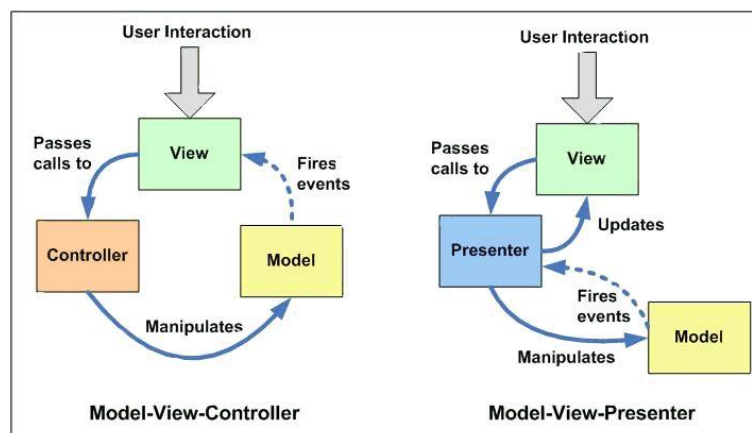



**Abbildung 39: *MVC* und *MVP Patterns***

Einerseits liefert der Einsatz von diesem Prinzip im Endeffekt ein weiteres Modularitätsniveau der App. Damit können die App-Erweiterung und die Testvorgänge der gesamten App erleichtert werden. Anderseits fordert der Einsatz viel mehr Kenntnisse beim Umgang mit der *Android*-Plattform, da nicht nur das Verstehen des Prinzips, sondern auch die eigentliche Implementierung

dieses Prinzips eine sehr umfangreiche und komplexe Herausforderung darstellt.

Als eine spätere Weiterentwicklung wäre die Zusammenfassung von ähnlichen *ListView*-Elementen in einem einzigen Element in Erwägung zu ziehen. Dabei könnte der Fall berücksichtigt werden, wenn mehrere *Nibo2*-Anwendungen mit den gleichen Projektnamen und Projekt-ID in der App initialisiert wurden.

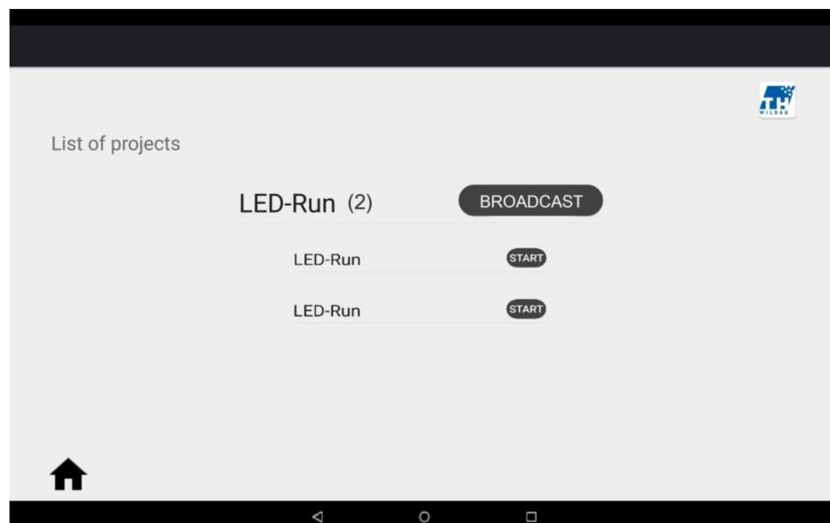



**Abbildung 40: Mögliche Weiterentwicklung**

Darüber hinaus könnte dadurch die funktionale Anforderung **F5-„Broadcast“** theoretisch vollständig realisiert werden. Hierbei könnten diese *Nibo*-Anwendungen zu einem *Button*-Objekt zusammengefasst werden und durch das Senden der einzigen Projekt-ID angesteuert werden.

Dafür könnte zum Beispiel die *Android*-Bibliothek *StickyListHeaders* eingesetzt werden. [21]

Weitere Entwicklungsmöglichkeiten der Anwendung können im Rahmen der Software-Projekte des Faches *C für eingebettete Systeme* entstehen.

# Abkürzungsverzeichnis

| | |
|---|---|
| **API** | Application Programming Interface |
| **App** | Application |
| **GUI** | Graphical User Interface |
| **ID** | Identification Number |
| **IR** | Infrared |
| **JSON** | JavaScript Object Notation |
| **LED** | Light-Emitting Diode |
| **MVP** | Model-View-Presenter Pattern |
| **SDK** | Software Development Kit |
| **SPI** | Serial Peripheral Interface |
| **UART1** | Universal Asynchronous Receiver Transmitter |
| **UML** | Unified Modeling Language |
| **VM** | Virtual Machine |
| **XML** | Extensible Markup Language |

# Glossar

| | |
|---|---|
| **Activity** | Ist einer der fundamentalen App-Bausteine der Android-Plattform. |
| **Array** | Ist eine Datenstruktur zur Speicherung mehreren Variablen mit einem Datentyp. |
| **Atmel** | Ist ein Hersteller von intergierten Schaltungen und Mikrocontrollern. |
| **Broadcast** | Bezeichnet eine Nachricht, die innerhalb eines Nachrichtennetzes an alle Teilnehmer dieses Netzes gesendet wird. |
| **Build-Prozesse** | Bezeichnen alle Erstellungsprozesse bei der automatischen Erzeugung eines Anwendungsprogramms. |
| **Code Listing** | Ist ein Abschnitt aus einem Anwendungsprogramm – Quellcode. |
| **Eclipse** | Ist eine quelloffene Entwicklungsumgebung. |
| **Exception** | Bezeichnet das Auftreten einer Fehlermeldung bei der Code-Ausführung eines Anwendungsprogramms. |
| **Framework** | Bezeichnet die Anwendungsstrukturrahmen, innerhalb derer diese Anwendung erstellt werden soll. |
| **Google I/O** | Ist eine Entwicklerkonferenz von *Google*. |
| **hand shake** | Ist ein automatisierter Prozess zur Einstellung des Datenflusses während eines |

Kommunikationsaufbaus zwischen zwei Systemen bevor der eigentliche Datenaustausch stattfindet.

**IntelliJ IDEA** Ist eine plattformunabhängige Entwicklungsumgebung von *JetBrains*.

**JetBrains** Ist ein tschechisches Software-Unternehmen, das die Entwicklungsumgebung *IntelliJ IDEA* entwickelt hat.

**Logging** Bezeichnet die Speicherung des Anwendungszustandes während des Betriebsablauf. Die dabei entstandenen Log-Daten werden später für den Debuggen-Vorgang der Anwendung verwendet.

**NAO** Ist ein humanoider Roboter, der von dem französischen Roboterhersteller *Aldebaran Robotics* entwickelt wurde.

**Nibo Serial Protocol** Ist ein Protokoll zum Datenaustausch zwischen den *Nibo2*-Robotern und einem PC.

**Nicai-Systems** Ist ein Roboterbausatz-Unternehmen, das solche Roboterbausätze entwickelt und produziert, wie zum Beispiel *Nibo2*, *NIBOBee* und *Nibo burger.*

**Parsing** Bezeichnet die Prozesse zum Eingabedatenlesen und zur späteren Umwandlung dieser Eingabedaten in geeigneteres Format.

**Pseudo-Code** Ist ein Programmcode, der nicht zur maschinellen Interpretation, sondern lediglich zur Veranschaulichung eines Paradigmas oder Algorithmus dient. [22]

**Socket** Bezeichnet einen Kommunikationsendpunkt in einem Datenfluss zwischen zwei Systemen.

| | |
|---|---|
| **Systemevent** | Bezeichnet eine Systeminformierung über ein eingetretenes System-Ereignis. |
| **Thread** | Ist ein leichtgewichtiger Auftragsträger. |
| **Ubuntu** | Ist eine kostenlose Linux Distribution. |
| **View Element** | Ist ein grundlegender Baustein der *Android*-Plattform für die Komponentenerstellung der *GUI.* |
| **XBee-Modul** | Ist eine Radiomodulart von *Digi International*. |

# Quellenverzeichnis


[1] Nibo Wiki. ***NIBO 2/Haupt-Controller*** [Abrufdatum 04.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/NIBO_2/Haupt-Controller

[2] Nibo Wiki. ***NIBO 2/Coprozessor*** [Abrufdatum 04.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/NIBO_2/Coprozessor

[3] Nibo Wiki. ***NIBO 2/Bodensensoren*** [Abrufdatum 04.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/NIBO_2/Bodensensoren

[4] Nibo Wiki. ***Distanzmessung*** [Abrufdatum 04.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/Distanzmessung

[5] Nibo Wiki. ***Grafikdisplay*** [Abrufdatum 05.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/Grafikdisplay

[6] Nibo Wiki. ***UCOM-IR2-X*** [Abrufdatum 05.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/UCOM-IR2-X

[7] Nibo Wiki. ***NDS3*** [Abrufdatum 07.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/NDS3

[8] Nibo Wiki. ***NXB2*** [Abrufdatum 07.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/NXB2

[9] Nibo Wiki. ***Nibo Serial Protocol*** [Abrufdatum 09.04.2017]
**URL**: http://www.nibo-roboter.de/wiki/Nibo_Serial_Protocol

[10] gamedev.ru. ***Синглтон (Singleton)*** [Abrufdatum 09.04.2017]
**URL**: http://www.gamedev.ru/code/terms/Singleton

[11] Android. ***Bluetooth*** [Abrufdatum 09.04.2017]
**URL**:https://developer.android.com/guide/topics/connectivity/bluetooth.html#ConnectionTechniques

[12] Android. ***Broadcasts*** [Abrufdatum 09.04.2017]
**URL**: https://developer.android.com/guide/components/broadcasts.html

[13] Anroid. ***List View*** [Abrufdatum 09.04.2017]
**URL**: https://developer.android.com/guide/topics/ui/layout/listview.html

[14] Android. **an*droid:bluetooth*** [Abrufdatum 09.04.2017]
**URL**:https://developer.android.com/reference/android/bluetooth/package-summary.html

[15] Stack Overflow Community. ***IOException: read failed, socket might closed - Bluetooth on Android 4.3*** [Abrufdatum 09.04.2017]
**URL**: http://stackoverflow.com/questions/18657427/ioexception-read-failed-socket-might-closed-bluetooth-on-android-4-3

[16] oracle.com. ***Class Class<T>*** [Abrufdatum 09.04.2017]
**URL**:https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getMethod-java.lang.String-java.lang.Class...-

[17] oracle.com. ***Invoking Methods*** [Abrufdatum 09.04.2017]
**URL**:https://docs.oracle.com/javase/tutorial/reflect/member/methodInvocation.html

[18] Stack Overflow Community. ***The application may be doing too much work on its main thread*** [Abrufdatum 09.04.2017]
**URL**: http://stackoverflow.com/questions/14678593/the-application-may-be-doing-too-much-work-on-its-main-thread

[19] codepath.com. ***Leveraging the Gson Library*** [Abrufdatum 09.04.2017]
**URL**: http://guides.codepath.com/android/leveraging-the-gson-library

[20] David Guerrero's blog. ***A brief introduction to a cleaner Android architecture: The MVP pattern*** [Abrufdatum 09.04.2017]
**URL**: https://davidguerrerodiaz.wordpress.com/2015/10/13/a-brief-introduction-to-a-cleaner-android-architecture-the-mvp-pattern/

[21] github.com. ***StickyListHeaders*** [Abrufdatum 09.04.2017]
**URL**: https://github.com/emilsjolander/StickyListHeaders

[22] Wikipedia.org. ***Pseudocode*** [Abrufdatum 11.04.2017]
**URL**: https://de.wikipedia.org/wiki/Pseudocode

# Abbildungsverzeichnis

**Abbildung 1: Nibo2** **Seite 7**

[https://cdn-reichelt.de/bilder/web/xxl_ws/X400/NIBO2_4.png (Abrufdatum 01.04.2017)]

**Abbildung 2: *CNY70-Sensoren*** **Seite 8**

[http://www.nibo-roboter.de/mediawiki/images/d/d7/Liniensensoren.jpg (Abrufdatum 01.04.2017)]

**Abbildung 3: Distanzsensor** **Seite 8**

[http://www.nibo-roboter.de/mediawiki/images/thumb/7/78/Distanz.jpg/681px-Distanz.jpg (Abrufdatum 01.04.2017)]

**Abbildung 4: Grafikdisplay** **Seite 9**

[http://www.nibo-roboter.de/mediawiki/images/thumb/7/78/Display.jpg/774px-Display.jpg (Abrufdatum 01.04.2017)]

**Abbildung 5: UCOM-IR2-X** **Seite 9**

[http://www.nibo-roboter.de/mediawiki/images/thumb/5/57/UCOM-IR2-X_b.jpg/800px-UCOM-IR2-X_b.jpg (Abrufdatum 01.04.2017)]

**Abbildung 6: NDS3- Erweiterungsmodul** **Seite 10**

[http://www.nibo-roboter.de/mediawiki/images/thumb/9/9b/NDS3_2.jpg/800px-NDS3_2.jpg (Abrufdatum 01.04.2017)]

**Abbildung 7: NXB2-Adapterplatine** **Seite 10**

[http://www.nicai-systems.com/images/stories/images/NXB2.jpg (Abrufdatum 01.04.2017)]

**Abbildung 8: Blue-Modul** **Seite 11**

[http://www.nicai-systems.com/images/stories/images/blue_modul.jpg (Abrufdatum 28.03.2017)]

**Abbildung 9: Samsung Galaxy NOTE 10.1 SM-P600** **Seite 11**

[http://images.samsung.com/is/image/samsung/ua-ru_SM-P6000ZWASEK_000215988_Front_black?$DT-Gallery$ (Abrufdatum 28.03.2017)]

**Abbildung 10: Android Studio Logo** **Seite 12**

[http://technicollit.com/wp-content/uploads/2016/04/android-studio-logo.png (Abrufdatum 28.03.2017)]

**Abbildung 11: VMWare Workstation Logo** **Seite 13**

[https://s3.amazonaws.com/neowin/news/images/uploaded/2014/10/vmwareplayer_medium.jpg (Abrufdatum 28.03.2017)]

**Abbildung 12: Systemarchitektur** **Seite 16**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 13: Datenpaket für den Datenaustausch** **Seite 18**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 14: Generiertes Datenpaket** **Seite 19**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 15: Ordnerstruktur des vorhandenen Projekts** **Seite 26**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 16: Ordnerstruktur des A2A-Moduls** **Seite 26**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 17: Ordnerstruktur des fertigen Projekts** **Seite 27**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 18: *Storyboard* – Beispiel** **Seite 29**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 19: *Storyboard* – Main Activity** **Seite 30**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 20: *Storyboard* – Device List Activity** **Seite 31**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 21: *Storyboard* – *Run Projects Activity*** **Seite 32**

Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 22: *Storyboard* – *Delete Projects Activity*** **Seite 33**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 09.04.2017)]

**Abbildung 23: *Storyboard* – Zusammenfassung** **Seite 34**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 24: Unerwarteter Output 1** **Seite 45**

[Foto: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 25: Unerwarteter Output 2** **Seite 45**

[Foto: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 26: Erwarteter Output** **Seite 46**

[Foto: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 27: Output mit einem Endsymbol** **Seite 46**

[Foto: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 28: `Nibo` und `NiboFactory` Klassen** **Seite 49**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 29: `ConnectThread` und `ConnectedThread` (*DeviceListActivity*)** **Seite 50**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 30: `ConnectThread` und `ConnectedThread` (*ProjectListActivity*)** **Seite 51**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 31: *MainActivity*** **Seite 52**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 32: *MainActivity* (mit ausgeblendeten Knöpfe)** **Seite 53**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 33: `activity_device_list.xml`** **Seite 55**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 34: `list_item_project_add.xml`** **Seite 55**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 35: *DeviceListActivity* (Screenshot)** **Seite 56**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 36: *ProjectListActivity* (Screenshot)** **Seite 58**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 37: *DeleteProjectActivity* (Screenshot)** **Seite 59**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 38: *JSON-Objekt*** **Seite 63**

[Screenshot: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

**Abbildung 39: *MVC* und *MVP Patterns*** **Seite 69**

[https://davidguerrerodiaz.files.wordpress.com/2015/10/18289.jpg (Abrufdatum 10.04.2017)]

**Abbildung 40: Mögliche Weiterentwicklung** **Seite 70**

[Zeichnung: Yerzhan Aitzhanov (Abrufdatum 10.04.2017)]

# Tabellenverzeichnis

**Tabelle 1: Funktionale Anforderungen** **Seite 5**

[Erstellt von Yerzhan Aitzhanov]

**Tabelle 2: Nicht-Funktionale Anforderungen** **Seite 5**

[Erstellt von Yerzhan Aitzhanov]

**Tabelle 3: Qualitätsanforderungen** **Seite 6**

[Erstellt von Yerzhan Aitzhanov]

# Diagrammverzeichnis

**Sequenzdiagramm 1: Initialisierungsphase** **Seite 20**

[Erstellt von Yerzhan Aitzhanov]

**Sequenzdiagramm 2: Starten und Stoppen einer *Nibo-Anwendung***

**Seite 21**

[Erstellt von Yerzhan Aitzhanov]

**Sequenzdiagramm 3: Starten und Stoppen zweier *Nibo-Anwendungen***

**Seite 23**

[Erstellt von Yerzhan Aitzhanov]

**Aktivitätsdiagramm 1: *Nibo-Teilsystem*** **Seite 24**

[Erstellt von Yerzhan Aitzhanov]

# *Code Listings*-Verzeichnis

**Code Listing 1: Inkludieren der Header-Datei** **Seite 27**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 2: Standby-Modus und die Initialisierung des Projekts**

**Seite 28**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 3: Starten und Stoppen des vorhandenen Projekts**

**Seite 28**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 4: `a2a_init()`- Methode** **Seite 37**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 5: `a2a_wait4StartMain()` - Methode** **Seite 39**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 6: `a2a_wait4ProjectId()` - Methode** **Seite 39**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 7: `main` -Methode des `Hello_Adapt2Android-LED` Beispielprojekts** **Seite 42**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 8: Erstellung eines `BluetoothSockets`** **Seite 57**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 9: Erstellung des `BluetoothSockets` (alte Variante)** **Seite 61**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 10: Umleiten in andere *Activity*** **Seite 64**

[Erstellt von Yerzhan Aitzhanov]

**Code Listing 11: Umleiten in andere *Activity* mit der Datenanheftung** **Seite 64**

[Erstellt von Yerzhan Aitzhanov]

# Beilagen

Auf der beigefügten DVD sind folgende Inhalte zu finden:

- **Dokument:**
  - Digitale Fassung der Bachelorarbeit.
- **Projektdateien:**
  - ZIP-Datei mit der *Android*-App,
  - ZIP-Datei mit der *Nibo*-Anwendung sowie den Beispielprojekten.