Design einer Bibliothek für die Kommunikation in einem Ad-hoc Multihopnetzwerk

Von
Hagen Wittlich
Matrikelnr. 561841

Bachelorarbeit zur Erlangung des Hochschulgrades
„Bachelor of Science“ (B.Sc.)
im Studiengang
Angewandte Informatik

1. Gutachter: Prof. Dr. Alexander Huhn
2. Gutachter: Prof. Dr.-Ing. Thomas Schwotzer

Eingereicht am 27. August 2019
Abstract (Deutsch)


Da diese Arbeit als grundlegendes Projekt und erster technischer Durchstich gedacht ist, auf dem später aufgebaut werden soll, stehen die Dokumentation, die Erweiterbarkeit der Bibliothek und die Sauberkeit des Codes im Fokus.
Abstract (Englisch)

The aim of this bachelor thesis is the development of a library for Android that acts as the foundation on which routing procedures for ad-hoc multihop networks can be implemented. The basic elements of such a network are supposed to be network hardware, which allow users to connect to them via smartphone and then act as a node in the network. Within this context, the library takes on the task of abstracting the communication between network hardware and Android applications, so that developers can focus on the communication between the nodes within the network. Abstracting the communication with the lower layers should enables a faster and more efficient development of android applications in this domain.

Since this paper and the developed android library function as a spike solution upon which future projects can be build, the author put the emphasis on clear documentation, expandability of the library and clean code.
# Inhaltsverzeichnis

Abstract (Deutsch).................................................................................................................. II
Abstract (Englisch).................................................................................................................. III
Abbildungsverzeichnis.............................................................................................................. VI
Abkürzungsverzeichnis.............................................................................................................. VII

1 Einleitung................................................................................................................................. 1
   1.1 Motivation .......................................................................................................................... 1
   1.2 Zielsetzung der Arbeit ....................................................................................................... 2
   1.3 Aufbau der Arbeit ............................................................................................................ 2

2 Grundlagen............................................................................................................................. 4
   2.1 Internet of Things ............................................................................................................. 4
   2.2 Drahtlose Ad-hoc Multihopnetzwerke ........................................................................... 4
   2.3 Observer Pattern ............................................................................................................. 5
   2.4 Template Method Pattern ............................................................................................. 6
   2.5 Decorator Pattern .......................................................................................................... 6
   2.6 Gerätekommunikation über AT-Befehle ....................................................................... 7
   2.7 Interfaces und abstrakte Klassen in Java ..................................................................... 8
   2.8 Erweiterbarkeit und Kopplung ..................................................................................... 9

3 Analyse................................................................................................................................... 11
   3.1 Zielgruppen ..................................................................................................................... 11
   3.2 Funktionale Anforderungen und nicht funktionale Anforderungen .......................... 11

4 Entwurf.................................................................................................................................. 13
   4.1 Grundlegende Komponenten ....................................................................................... 13
   4.2 Erweiterung der Komponenten um Protocol ............................................................... 16
   4.3 Entwurf der notwendigen Klassen .............................................................................. 19
      4.3.1 Klassendiagramm der Grundstruktur ................................................................. 19
      4.3.2 AbstractSerialConnection ............................................................................... 20
      4.3.3 SerialInputListener ............................................................................................ 22
      4.3.4 AbstractDevice .................................................................................................... 23
      4.3.5 DeviceListener ..................................................................................................... 24
Abbildungsverzeichnis

Abbildung 1: Vereinfachte Darstellung der Kommunikationsflüsse in einem Netzwerk............ 1
Abbildung 2: Systemdarstellung der Nutzung von AT-Befehlen ........................................... 8
Abbildung 3: Einfache schematische Darstellung der zu entwickelnden Lösung .................. 14
Abbildung 4: UML-Diagramm der grundlegenden Komponenten der Bibliothek .............. 16
Abbildung 5: Komponentendiagramm mit Protokollerweiterung .................................. 16
Abbildung 6: Erweiterte schematische Darstellung der Abstraktionsebenen .................. 18
Abbildung 7: UML-Klassendiagramm der grundlegenden Bibliotheksstruktur .................. 19
Abbildung 8: UML-Klassendiagramm mit Subklassen ...................................................... 31
Abbildung 9: Schematischer Testaufbau ............................................................................ 37
Abbildung 10: Grafische Nutzeroberfläche für Testszenario 1 ......................................... 38
Abbildung 11: Grafische Nutzeroberfläche für Testszenario 2 ......................................... 41
Abbildung 12: Grafische Nutzeroberfläche für Testszenario 3 ......................................... 44
Abkürzungsverzeichnis

<table>
<thead>
<tr>
<th>Abkürzung</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Code für “ATTENTION”</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communications</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>USB OTG</td>
<td>Universal Serial Bus On-The-Go</td>
</tr>
<tr>
<td>TTL</td>
<td>Transistor-Transistor Logic</td>
</tr>
</tbody>
</table>
1 Einleitung

1.1 Motivation

Drahtlose Kommunikation hat in der heutigen Gesellschaft eine außerordentliche Stellung inne. Als ein wichtiger Pfeiler des Informationsaustauschs und der Datenübertragung ist sie ein bedeutsamer Faktor in digitalen Gesellschaften. Neben großflächig verfügbarem WLAN und der Datenübertragung in Mobilfunknetzen haben durch die rasante Entwicklung des Internet of Things (IoT) zahlreiche neue drahtlose Kommunikationstechnologien an Bedeutung gewonnen. Diese erfüllen Anforderungen, die sich über etablierte Infrastrukturen schlecht oder nicht abbilden lassen. Durch diese Technologien steigt die Bedeutung von drahtlosen ad-hoc Netzen, die vom Internet und Mobilfunknetzen unabgedeckte Bereiche abdecken oder eigenständig und vom Internet unabhängig für die Kommunikation genutzt werden können. Um solche Netzwerke zu realisieren, gibt es unterschiedliche Technologien und Geräte. Für Entwickler(innen) ist es mitunter sehr aufwändig, die Kommunikation zwischen ihrer Applikation und entsprechender Netzwerkhardware zu implementieren, da es notwendig ist, die spezifischen Eigenschaften des Netzwerkgerätes zu kennen, um es zu steuern.

Abbildung 1 stellt schematisch die typischen Kommunikationspfade zwischen einer Applikation und der Netzwerkhardware in einem Netzwerk dar:

\[ \text{Abbildung 1: Vereinfachte Darstellung der Kommunikationsflüsse in einem Netzwerk} \]

Die Kommunikation zwischen den Netzwerkteilnehmern auf der Applikationsschicht wird durch von den Entwickler(innen) festgelegte Regeln durchgeführt. Diese Kommunikationslogik ist idealerweise unabhängig von der darunterliegenden Netzwerkhardware. Da jedoch die realen Kommunikationsflüsse über die Routen 1 und 2 verlaufen, und 3 nur den logischen Kommunikationsfluss über die Routen 1 und 2 verlaufen, und 3 nur den logischen Kommunikationsfluss
darstellt, müssen Applikationsentwickler(innen) auch mit diesen beiden Kommunikationsflüssen vertraut sein. Dabei kann das Aufbauen der Verbindung zwischen der Applikation und dem Netzwerkgerät eine komplexe Aufgabe sein, die es erfordert, gründliche Kenntnisse über die Netzwerkhardware zu haben. Dementgegen ist der Aufbau einer Kommunikation von zwei kompatiblen Netzwerkgeräten oft durch Standards, zugrundeliegende Technologien oder die Hersteller geregelt, sodass Entwickler(innen) diesem Punkt (2) weniger Aufmerksamkeit schenken müssen.

1.2 Zielsetzung der Arbeit

Im Rahmen dieser Arbeit soll eine Androidbibliothek der Programmiersprache Java entworfen werden, die dafür sorgt, dass Entwickler(innen) leichter Android Applikationen mit Bezug zu ad-hoc Multihopnetzwerken erstellen können. Wie im vorherigen Abschnitt gezeigt, sollen sie sich auf die logische Kommunikation zwischen den Applikationen konzentrieren können. Der Kommunikationsfluss zwischen der Applikation und der Netzwerkhardware soll durch die Bibliothek unterstützt werden. Die zu entwerfende Lösung stellt eine Abstractionsschicht zwischen der Hardware und der Software dar, die Entwickler(innen) einen Satz an sinnvollen Methoden bieten soll, mithilfe derer sie neue Netzwerkapplikationen schreiben können.


1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in mehrere Teile gegliedert. Im ersten Teil der Arbeit, dem Kapitel „Grundlagen“, sollen Begrifflichkeiten und grundlegendes Wissen dargestellt werden, die für das Verständnis hilfreich oder notwendig sind.

Das Kapitel „Analyse“ zeigt entsprechende funktionale und nicht-funktionale Anforderungen auf, die im Rahmen der zu entwickelnden Lösung berücksichtigt
werden sollen. Hierbei konzentriert sich der Autor auf die wichtigsten Anforderungen.

Im Teil „Entwurf“ wird schließlich das Konzept der zu entwickelnden Lösung dargestellt. Der Aufbau der Lösung wird dargelegt und die in der Bibliothek enthaltenen Klassen beschrieben. Ebenso werden wichtige Entwurfsentscheidungen diskutiert und begründet.

Die Umsetzung der Lösung sowie die Herausforderungen, die dabei aufraten, werden im Kapitel „Implementierung“ beschrieben.


Das Kapitel „Ausblick“ zeigt weiterführende Fragestellungen und Thematiken auf, die über den Rahmen dieser Arbeit hinausgehen.
2 Grundlagen

2.1 Internet of Things

Das Internet of Things oder Internet der Dinge bezeichnet die Vernetzung von physischen Objekten, auf die über das Internet zugegriffen werden kann. Die Objekte sind mit der entsprechenden Technologie gekoppelt, sodass sie kommunizieren und über interne oder externe Zustände Informationen aufnehmen können. [1]

Aufgrund der Anforderungen des IoT haben sich rasant neue drahtlose Netzwerktechnologien entwickelt, die für verschiedene Bereiche wie Sensornetzwerke oder auch im Smart Home Bereich Anwendung finden. Beispiele hierfür sind Zigbee, Z-Wave, LoRa, SigFox oder auch Bluetooth Low Energy. [2]

Das IoT hat sich nicht zuletzt auch aufgrund von wirtschaftlichen und technologischen Neuerungen entwickeln können. Einige Beispiele dafür [1]:

- Günstige Sensoren, Bandbreite, Speicher- und Prozessortechnologien ermöglichen die Wirtschaftlichkeit von kommerziellen Lösungen
- Smartphones als allgegenwärtige Begleiter ermöglichen Interaktion und Steuerung von IoT Objekten über das Internet
- Umfassende Verfügbarkeit von WLAN als eine Schlüsseltechnologie

Für die vorliegende Arbeit hat das Internet of Things in der Hinsicht Relevanz, als dass die in diesem Rahmen entworfene Lösung für IoT Anwendungsfälle eingesetzt werden kann und viele der Entwicklungen des IoT die vorliegende Arbeit beeinflusst haben.

2.2 Drahtlose Ad-hoc Multihopnetzwerke

Die vorliegende Arbeit soll als Grundlage dafür dienen, drahtlose ad-hoc Multihopnetzwerke über Android Applikationen aufzubauen. Daher soll an dieser Stelle näher darauf eingegangen werden, was drahtlose ad-hoc Multihopnetzwerke sind und wofür sie typischerweise eingesetzt werden.
2. Grundlagen


Ein weiteres Merkmal solcher Netzwerke ist häufig das Fehlen einer festen Infrastruktur, was für gewöhnlich mit einer erhöhten Kommunikationskomplexität einhergeht [4]. Zusätzlich wird die Komplexität dadurch beeinträchtigt, dass sich schnell Veränderungen in der Netzwerktopologie ergeben können [5]. Ein weiterer Nachteil solcher Netze ist eine hohe Anfälligkeit für Sicherheitsangriffe, die auf ihre offene und verteilte Natur zurückzuführen ist [4].

Anwendungsgebiete solcher Netzwerke können beispielsweise Sensornetzwerke sein; ebenso können auch Meshnetzwerke zur Erweiterung der Reichweite von Internetzugangspunkten darüber abgebildet werden. Zusätzlich können drahtlose Multihopnetzwerke für die Kommunikation eingesetzt werden, wenn keine feste Infrastruktur wie das Internet oder Mobilfunknetze zur Verfügung steht - wie beispielsweise in Katastrophenfällen - oder wenn sie bewusst umgangen werden soll. [3]

2.3 Observer Pattern


Prinzipiell funktioniert das Observer Pattern wie folgt: Es gibt ein zu beobachtendes Objekt und einen oder mehrere Beobachter. Die Beobachter registrieren sich beim beobachteten Objekt in einer Liste. Bei Änderungen im beobachteten Objekt ruft dieses eine Methode für sämtliche registrierte Beobachter in seiner Liste auf, die dazu führt, dass diese auf die Veränderung reagieren.
Das Observer Pattern eignet sich in verschiedenen Situationen [7]:

- Falls eine abstrakte Komponente in zwei Teile geteilt werden kann, bei dem der eine Teil vom anderen abhängig ist. Das Kapseln dieser zwei Teile in unterschiedliche Objekte verringert den Grad der Kopplung und ermöglicht es, dass sie unabhängig voneinander geändert und wiederverwendet werden können.
- Wenn die Änderung an einem Objekt das Ändern von einem oder mehreren anderen Objekten nach sich zieht, deren genaue Anzahl unbekannt ist.
- Falls ein betrachtetes Objekt andere Objekte benachrichtigen muss, ohne vorher zu wissen, welche Objekte das sind.

2.4 Template Method Pattern


Zur Veranschaulichung soll folgendes Beispiel dienen:
Unter die abstrakten, sequenziellen Prozessschritte „Zutaten kaufen“, „Essen zubereiten“ und „Essen servieren“ lassen sich sowohl das Zubereiten einer Tiefkühlpizza, als auch das Zubereiten eines Gerichts in einem Restaurant einordnen. Die abstrakten Schritte sind gleich, in der Implementierung sind sie jedoch verschieden.

Das Template Method Pattern gibt also die Struktur eines Prozesses vor, überlässt jedoch bestimmte Schritte den entsprechenden Subklassen zur Implementierung. Es wird in Java überwiegend in abstrakten Klassen angewandt. [8]

2.5 Decorator Pattern

Das Decorator Pattern ist ebenfalls eines der klassischen Entwurfsmuster der Gang of Four. Dies ermöglicht es, einem Objekt zur Laufzeit neue Verantwortlichkeiten


2.6 Gerätekommunikation über AT-Befehle

Die Kommunikation mit Netzwerkgeräten kann prinzipiell auf vielen unterschiedlichen Wegen erfolgen. Je nachdem, welchen Weg der Hersteller vorgesehen hat, muss die softwareseitige Kommunikation angepasst werden. Einige Hersteller arbeiten dabei mit proprietären, andere mit standardisierten Protokollen für die Kommunikation; wieder andere erweitern standardisierte Protokolle, um dadurch die eigenen Bedürfnisse zu erfüllen [10].


Anfänglich wurde der AT-Befehlssatz für den Einsatz in analogen Modems entwickelt. 1996 wurde dieser Befehlssatz als Grundlage für den Global System for Mobile Communications (GSM) Standard 07.07, *AT Command Set for GSM Mobile*
2. Grundlagen

*Equipment*, genutzt und ist heute im Telekommunikations- und Netzwerkbereich verbreitet. [10]


Abbildung 2: Systemdarstellung der Nutzung von AT-Befehlen

2.7 Interfaces und abstrakte Klassen in Java

Für den Entwurf einer angemessenen und durch Entwickler(innen) erweiterbaren Lösung ist es wichtig, an geeigneten Stellen auf Interfaces beziehungsweise abstrakte Klassen zurückzugreifen. Diese spezifizieren verbindlich, welche Bestandteile und Methoden Klassen haben müssen, die die Lösung erweitern. An dieser Stelle soll deshalb auf Interfaces und abstrakte Klassen in Java eingegangen werden, da in der Entwurfsphase entschieden werden muss, wann sie angewandt werden sollen.
Generell stellt Java mit beiden Mechanismen jeweils eine Möglichkeit zur Verfügung um einen Typen zu definieren, der unterschiedliche Implementierungen erlaubt. Abstrakte Klassen haben gegenüber Interfaces die Einschränkung, dass sie von der Mehrfachvererbung ausgeschlossen sind. [12]


Sowohl Interfaces als auch abstrakte Klassen können einer Klasse vorschreiben, welche Methoden sie implementieren muss. Während in Interfaces die Deklarierung der Methode hierzu ausreicht, wird in abstrakten Klassen hierzu das Schlüsselwort `abstract` verwendet. In abstrakten Klassen ist es möglich, Methoden direkt zu implementieren, die der Subklasse zur Verfügung stehen. Seit Java 8 können auch in Interfaces über das Schlüsselwort `default` Methodenkörper implementiert werden. Während sämtliche Methoden von Interfaces implizit `public` sind, bieten abstrakte Klassen die Möglichkeit, den Methodenzugriff auch als `private`, `protected` oder ohne Schlüsselwort (package) festzulegen [9].

Zusammenfassend kann festgehalten werden, dass Interfaces generell aufgrund der Möglichkeit der Mehrfachvererbung vorzuziehen sind, wenn es nur um die Vererbung beziehungsweise das Vorschreiben von Methoden geht. Zusätzlich sind sie leichter austauschbar, da sie nicht tiefer in die Struktur von Klassen eingreifen. Abstrakte Klassen sind hingegen in Situationen besser geeignet, in denen es gewollt ist, bestimmte Strukturen vorzugeben, die in all ihren Subklassen identisch sind oder sein sollen. Beide erfüllen also unterschiedliche Aufgaben.

## 2.8 Erweiterbarkeit und Kopplung

Eines der Ziele der vorliegenden Arbeit ist es, eine grundlegende Lösung zu erarbeiten, die Funktionalitäten für den vorliegenden Typ von Netzwerkgeräten sowie eine serielle Verbindung implementiert. Da der Rahmen der Arbeit stark
2. Grundlagen

begrenzt ist, wird als Lösung eine Bibliothek erwartet, die leicht um Funktionalitäten für weitere Netzwerkgeräte und Arten von seriellen Verbindungen erweitert werden kann.

Erweiterbarkeit bezeichnet die Fähigkeit eines Softwaresystems, neu hinzugefügte Elemente oder Funktionalitäten aufzunehmen, ohne dass es in wesentlichen Aspekten geändert werden muss. Ein Softwaresystem besitzt ein hohes Maß an Erweiterbarkeit, wenn neue Objekte hinzugefügt werden können, ohne dass an bestehenden Klassen etwas geändert werden muss. Erweiterbare Software hat potentiell eine erhöhte Lebensdauer, kann jedoch eine negative Auswirkung auf die Softwareeffizienz haben. [13]


Der Einsatz von Interfaces und abstrakten Klassen ist ein adäquates Mittel, um die Anzahl der Abhängigkeiten zwischen Softwareeinheiten zu verringern und ihre Erweiterbarkeit zu erhöhen. [15]
3 Analyse

3.1 Zielgruppen

Die zu entwickelnde Bibliothek richtet sich im Wesentlichen an zwei Zielgruppen:
1. Entwickler(innen), die die Bibliothek erweitern wollen
2. Entwickler(innen), die die Funktionen der Bibliothek zur Entwicklung von
   Applikationen nutzen wollen

Für die erste Zielgruppe sind verschiedene Aspekte besonders relevant: Die Bibliothek soll leicht erweiterbar und die Klassen für Netzwerkgeräte und Verbindungstypen austauschbar sein. Eine enge Kopplung der einzelnen Komponenten soll deshalb weitestgehend vermieden werden. Dafür benötigt die Bibliothek eine geeignete Struktur, die durch die Verwendung von Interfaces und Abstrakten Klassen möglichst modular gehalten werden soll.


Die Anforderungen beider Zielgruppen sollen bei der Umsetzung der Lösung berücksichtigt werden.

3.2 Funktionale und nicht-funktionale Anforderungen

Für einen ersten technischen Durchstich soll die Bibliothek es ermöglichen, über Bluetooth eine serielle Verbindung zur Netzwerkhardware aufzubauen. Hierbei soll der Nutzer der Bibliothek lediglich die MAC-Adresse des Netzwerkgerätes spezifizieren müssen, den Rest soll die Bibliothek erledigen können.

Weiterhin soll die Bibliothek Möglichkeiten bereitstellen, Nachrichten an das ad-hoc Netzwerk über das Netzwerkgerät zu senden. Nachrichten sollen als simple Strings an die Bibliothek übergeben werden können.


Zusätzlich sollen eventuell auftretende Fehler bis zur Ebene der Nutzer(innen) weitergeleitet werden, sofern eine Fehlerbehandlung nicht schon vorher sinnvoll ist. Klassen und Schnittstellen müssen dementsprechend gestaltet werden.

Die im Rahmen dieser Arbeit zu erstellende Lösung soll unter Beachtung von nichtfunktionalen Anforderungen erstellt werden. Dazu soll sie einen hohen Grad an Erweiterbarkeit sowie möglichst geringe Kopplung aufweisen. Dies soll es ermöglichen, die Bibliothek zukünftig um weitere Funktionalitäten zu erweitern, die beispielsweise den Aufbau der seriellen Verbindung über USB erlauben. Weiterhin soll die Struktur der Bibliothek so gestaltet sein, dass das Gerät austauschbar sein kann, das als Einstiegspunkt für das ad-hoc Netzwerk dient.
4 Entwurf

4.1 Grundlegende Komponenten


Grundsätzlich ist es erstrebenswert, mit wenigen und einfachen Befehlen die Funktionen der Bibliothek zu nutzen. Ein idealer Aufruf der Funktionen der Bibliothek könnte wie folgt aussehen:

```java
Device device = new Device(new Connection(targetAddress));
device.connect();
device.send(„hello world“);
```

Die drei Zeilen Code müssen dementsprechend die gesamte Arbeit leisten, die notwendig ist, um eine Verbindung mit dem Netzwerkgerät herzustellen und den Text „hello world“ an andere Netzwerkteilnehmer zu senden.

Das Connection-Objekt spezifiziert, wie die Verbindung aufgebaut wird, beispielsweise per Bluetooth. Die spezifische Connection-Klasse muss dann alle notwendigen Parameter enthalten, die für den Verbindungsaufbau notwendig sind. Im Falle von Bluetooth könnte das die MAC-Adresse sein, im Falle einer USB Verbindung zum Beispiel der entsprechende Port.

Die Connection-Komponente bildet also die grundlegende Schicht, über die mit dem physischen Netzwerkgerät kommuniziert wird. Die Device-Komponente bildet eine Abstraktion der Eigenschaften und Fähigkeiten des Netzwerkgeräts, die Entwickler(innen) Möglichkeiten bieten, mit dem Gerät zu interagieren ohne seinen internen Aufbau zu kennen. Abbildung 3 zeigt dies schematisch:

Abbildung 3: Einfache schematische Darstellung der zu entwickelnden Lösung

Die beiden Komponenten Device und Connection müssen miteinander kommunizieren können. Eingehende Nachrichten vom Netzwerkgerät werden über die serielle Verbindung von der Connection-Komponente gelesen, und müssen für die letztendliche Applikation aufbereitet und von der eventuell vorhandenen speziellen Codierung des Netzwerkgeräts bereinigt werden. Im Folgenden soll verdeutlicht werden, was damit gemeint ist.

Das für diese Arbeit vorliegende Netzwerkgerät codiert eingehende Nachrichten vom Netzwerk beispielsweise mit einem entsprechenden Präfix, das mit den Buchstaben „LR“ beginnt, gefolgt von der Quelladresse als Hexadezimalzahl und der Länge der Nachricht als Hexadezimalzahl. So würde eine Nachricht „hello“ von einem anderen Netzwerkgerät auf der seriellen Schnittstelle zum Beispiel wie folgt aussehen:
4. Entwurf

LR,FFCA,05,hello

Für Entwickler(innen) ist hierbei die Nachricht „hello“ das Wichtige und je nachdem, wie auf der Ebene der Applikation mit Adressen umgegangen wird, eventuell noch die Adresse des Absendergerätes.


In der vorliegenden Arbeit benachrichtigt die Connection-Komponente jedes Mal ihren Observer, ein Device-Objekt, wenn eine neue Nachricht über die serielle Schnittstelle eingeht. Der Observer muss ein spezielles Interface (SerialInputListener) implementieren, welches von der Connection-Komponente gefordert wird.

Ein Device-Objekt kann sich als SerialInputListener (Observer) bei einem Connection-Objekt registrieren. Dadurch erlangt das Connection-Objekt Zugriff auf die im SerialInputListener Interface spezifizierten Methoden, und kann Nachrichten an das Device-Objekt weiterleiten. Ein Beispiel hierfür wäre der folgende Code Aufruf, der im Connection-Objekt getätigt werden könnte, sobald eine eingehende Nachricht „message“ erhalten wurde:

```java
serialInputListener.onSerialMessageReceived(message);
```

Hier ruft das Connection-Objekt die Methode `onSerialMessageReceived(String)` auf, die im Device-Objekt durch das Interface implementiert sein muss. Das Device-Objekt kann in seiner Rolle als SerialInputListener dann auf die eingehende Nachricht reagieren und sie entsprechend aufbereiten.
4. Entwurf

Im nächsten Schritt muss die Nachricht dann der Applikation bereitgestellt werden. Hier bietet sich genau das gleiche Muster an. Die Applikation kann eine Klasse zur Verfügung stellen, die ein spezielles DeviceListener Interface implementiert. Ein Objekt dieser Klasse kann sich dann als DeviceListener in einem Device-Objekt registrieren. Dies macht es möglich, fortan Nachrichten zu erhalten, die von der seriellen Schnittstelle im Connection-Objekt kommen und durch das Device-Objekt für die Applikation entsprechend aufbereitet/bereinigt worden sind. Die beiden Komponenten und die beiden Interfaces sind Bestandteile der zu entwickelnden Lösung, die grafisch in Abbildung 4 in der Unified Modeling Language (UML) dargestellt sind:

Abbildung 4: UML-Diagramm der grundlegenden Komponenten der Bibliothek

4.2 Erweiterung der Komponenten um Protocol

Die im Rahmen dieser Arbeit entwickelte Bibliothek wurde dazu entwickelt, Entwickler(innen) die Möglichkeit zu geben, mit einfachen Schritten ein eigenes Kommunikationsprotokoll innerhalb eines ad-hoc Multihopnetzwerkes aufzusetzen. Um dies zu bewerkstelligen, sollen die in Abschnitt 4.1 gezeigten Komponenten wie in Abbildung 5 erweitert werden:

Abbildung 5: Komponentendiagramm mit Protokollerweiterung


Die in Abschnitt 4.1 beschriebene ideale Nutzung der Bibliothek ändert sich hier entsprechend:

```java
Protocol protocol = new Protocol(new Device(new Connection(targetAddress)));
protocol.connect();
protocol.sendBroadcast(“hello world”);
```

Im obigen Beispiel wurde die Methode send() des Gerätes durch die Methode sendBroadcast() eines fiktiven Protokolls ersetzt, die eine Nachricht an sämtliche Netzwerkteilnehmer schicken soll, auch über mehrere Hops hinweg. Die

---

tatsächlichen Methoden, die das Protocol-Objekt bietet, hängen von der entsprechenden Implementierung durch die Entwickler(innen) ab.

Für diese Erweiterung soll die Bibliothek eine abstrakte Basisklasse für Protokollimplementatioen bereitstellen, ebenso wie ein zusätzliches ProtocolListener Interface, das von anderen Klassen implementiert werden kann um entsprechend Nachrichten zu erhalten.


![Abbildung 6: Erweiterte schematische Darstellung der Abstraktionsebenen](image)

Anfänglich hat der Autor erwogen, die drei Komponenten über das Decorator Pattern zu verbinden, und die Connection-Komponente sukzessive darüber mit Device- und Protocol-Funktionalitäten zu dekorieren. Der Grundgedanke war der, dass sich die drei Komponenten in gewissen Funktionalitäten überschneiden und eventuell über dieses Entwurfsmuster abgebildet werden können. Dies stellte sich jedoch speziell hinsichtlich der Umsetzung in Java als nicht praktikabel heraus, ohne enge Kopplungen und eine unübersichtliche Struktur zu schaffen. Besonders die Vielfalt und die voraussichtlich starken Unterschiede zwischen Netzwerkgeräten und Protokollen macht es schwer, hier angemessene Strukturen zu entwerfen, die sich mit diesem Entwurfsmuster sinnvoll vereinfachen lassen.
4. Entwurf

4.3 Entwurf der notwendigen Klassen

4.3.1 Klassendiagramm der Grundstruktur

Nachdem in den vorherigen Abschnitten die Komponenten der Bibliothek erklärt wurden, soll in diesem Abschnitt der Entwurf konkretisiert werden. Dazu werden die Komponenten in Klassen spezifiziert.

Abbildung 7 zeigt die grundlegende Struktur der Bibliothek als UML-Klassendiagramm. Im Vordergrund stehen hier die abstrakten Klassen und Interfaces, die bereits in den vorhergehenden Teilen grob beschrieben wurden. Setter- und Gettermethoden für die Protected-Attribute der einzelnen Klassen (mit der UML Notation ’#’) wurden aus Gründen der Übersichtlichkeit in der Abbildung nicht berücksichtigt.

Abbildung 7: UML-Klassendiagramm der grundlegenden Bibliotheksstruktur
4. Entwurf


4.3.2 AbstractSerialConnection


Innerhalb der Bibliothek dient diese Klasse als Schablone für entsprechende Subklassen, die eine konkrete Art der Verbindung herstellen sollen, beispielsweise über Bluetooth oder USB. Im Rahmen der vorliegenden Arbeit wird von dieser Basisklasse eine nicht-abstrakte Subklasse implementiert, die eine serielle Verbindung zum Netzwerkgerät via Bluetooth aufbauen kann. Die Klasse AbstractSerialConnection soll dabei als Basisklasse alle notwendigen Attribute und Methoden beinhalten, die ihre Subklassen gemeinsam haben müssen. Dabei gilt es, sinnvoll notwendige Strukturen vorzugeben ohne unnötig einzuschränken.

Um den in 4.1 und 4.2 beschriebenen Kommunikationsfluss zwischen den Komponenten Connection und Device zu ermöglichen, sollen alle Subklassen dieser Klasse ein Attribut der Klasse SerialInputListener beinhalten, das in der Basisklasse implementiert werden soll. Hierin liegt die Begründung, weshalb an dieser Stelle eine abstrakte Klasse gewählt wurde, und kein Interface. Die geringste Sichtbarkeit, die für das Attribut gewählt werden soll, ist „Protected“. Somit können Subklassen auch ohne Setter- und Gettermethoden auf das Attribut zugreifen. Um
4. Entwurf

auch von außen auf das Attribut zugreifen zu können, werden zusätzlich eine Setter- und eine Gettermethode implementiert.

Schließlich gibt es Methoden, die jede Subklasse von `AbstractSerialConnection` anbieten können soll, und auf deren Existenz sich Entwickler(innen) verlassen können müssen. Es muss möglich sein, über eine Methode eine Verbindung zum Netzwerkgerät aufzubauen. Ebenso soll eine Verbindung beendet werden können. Die eingangs genannte Funktionsfähigkeit, Daten zu empfangen und zu senden, muss ebenfalls abbildbar sein. Um diese Vorgaben zu erreichen, weist die Klasse `AbstractSerialConnection` wie in Abbildung 7 drei abstrakte Methoden auf, die von ihren Subklassen implementiert werden müssen:

- `connect()` – baut die Verbindung zum Netzwerkgerät auf
- `disconnect()` – beendet die Verbindung zum Netzwerkgerät
- `write(byte[])` – schreibt Daten in die serielle Schnittstelle

Damit sind jedoch nur drei der vier Vorgaben erreicht. Das Empfangen von Daten muss in den entsprechenden Subklassen gelöst werden und soll nicht von außen über eine Methode erreichbar sein. Stattdessen werden hier Methoden des Interfaces `SerialInputListener` eingesetzt, um eingehende Nachrichten an entsprechende Implementierungen der Schnittstelle weiterzuleiten.

Für die `connect()` und `disconnect()` Methoden hat der Autor in Betracht gezogen, das Template Method Pattern zu nutzen. Prinzipiell können die beiden Funktionen jeweils in drei Phasen aufgeteilt werden:

- Schritte, die vor dem Verbindungsaufbau/-abbau stattfinden
- den Verbindungsaufbau/-abbau
- Schritte, die anschließend stattfinden sollen

Dementsprechend wären `connect()` und `disconnect()` finale Methoden und es gäbe zusätzliche abstrakte Methoden, die implementiert werden müssten und dann in entsprechender Reihenfolge aufgerufen werden würden. Die Anwendung des Template Method Pattern ist hier jedoch aus aktueller Sicht nicht sinnvoll, da diese Schritte häufig nicht notwendig sind und die Reihenfolge nicht zwingend
4. Entwurf

vorgegeben werden muss. An diesem Punkt würde unnötig Struktur vorgegeben werden, ohne dass der Autor ihre Nutzung antizipieren kann.

4.3.3 SerialInputListener

Das Interface SerialInputListener ist das Bindeglied zwischen der Connection- und der Device-Komponente. Über das Interface wird die Funktionalität des Empfangs von Nachrichten abstrahiert. Innerhalb der Subklasse von AbstractSerialConnection kann das Interface aufgerufen werden, da es als Attribut vorhanden ist. Das Interface schreibt allen seinen Implementierungen vor, über die Funktion onSerialDataReceived(byte[]) Nachrichten zu empfangen. Weiterhin sollen Subklassen der AbstractSerialConnection die Möglichkeit haben, Fehler an den entsprechenden SerialInputListener weiterzureichen, weshalb das Interface eine Methode onError(Exception) aufweist.


Diese drei Methoden können von den Subklassen der AbstractSerialConnection direkt aufgerufen werden, wenn das Attribut serialInputListener bei ihnen gesetzt ist. Werden über die serielle Verbindung Daten empfangen, so kann die entsprechende Connection Klasse die Daten mit serialInputListener.onSerialDataReceived(byte[]) entsprechend weiterleiten.

Für die Rolle des SerialInputListener sind zwar die Klassen aus der Device Komponente gedacht, jedoch kann beispielsweise auch eine Android Activity Klasse das Interface implementieren und sich in einer Subklasse der AbstractSerialConnection als SerialInputListener setzen. Hierbei ist zu beachten, dass Entwickler(innen) sich in diesem Fall jedoch noch immer mit den spezifischen Eigenschaften des Netzwerkgerätes auseinandersetzen müssen. Arbeitet das Gerät beispielsweise mit AT Steuerbefehlen, so müssen diese bekannt sein, um es sinnvoll
einzusetzen. Des Weiteren müssen sich die Entwickler(innen) damit beschäftigen, eingehende Nachrichten eventuell zu entpacken, um an die enthaltenen Daten zu gelangen.

4.3.4 AbstractDevice


Sämtliche Subklassen sollen die folgenden Methoden anbieten:

- `connect()` – ruft die connect() Methode des SerialConnection Objektes auf und führt gerätspezifische Prozesse aus, die in diesem Rahmen notwendig sind
- `disconnect()` – ruft die disconnect() Methode des SerialConnection Objektes auf und führt gerätspezifische Prozesse aus, die in diesem Rahmen notwendig sind
- `sendDeviceCommand(String)` – sendet einen Befehl an das Netzwerkgerät. Diese Methode bietet Entwicklern und Entwicklerinnen mehr Möglichkeiten, mit dem Netzwerkgerät zu interagieren, erfordert jedoch auch spezifisches Wissen darüber
- `send(String)` – sendet eine Nachricht an andere Netzwerkteilnehmer
Methoden, die über die oben genannten hinausgehen, müssen in den entsprechenden Subklassen implementiert werden. Ein Beispiel hierfür wäre eine `configure()` Methode, die spezifische Einstellungen am Netzwerkgerät vornimmt, wie zum Beispiel die Empfangs- und Sendefrequenz.

### 4.3.5 DeviceListener


Das `DeviceListener` Interface ähnelt dem `SerialInputListener` Interface, da es ebenfalls ein Verbindungsstück für zwei Abstraktionsschichten ist. Die Methoden `onError()` und `onConnect()` sind deshalb auch hier zu finden. Nachdem die für die entsprechende Schicht spezifischen Maßnahmen durchgeführt wurden (zum Beispiel das Behandeln oder Maskieren eines Fehlers bei `onError()` oder zusätzliche Einstellungen bei `onConnect()`), kann die Kommunikation an die nächsthöhere Schicht weitergereicht werden.

 Anders als beim `SerialInputListener` Interface, dass nur die Methode `onSerialDataReceived(byte[])` enthält um eingehende Nachrichten an die Device Schicht weiterzuleiten, stellt das `DeviceListener` Interface hierfür zwei Methoden zur Verfügung. Der Grund dafür ist, dass in der entsprechenden Subklasse von `AbstractDevice` bereits entschieden werden soll, ob eingehende Nachrichten von anderen Netzwerkteilnehmern stammen oder ob sie Rückmeldungen vom Netzwerkgerät sind. Die entsprechenden Methoden `onNetworkMessageReceived(String)` und `onDeviceMessageReceived(String)` erlauben der Implementation des `DeviceListener` Interfaces, entsprechend
unterschiedlich mit diesen beiden Fällen umzugehen. Für die innerhalb dieser Arbeit entworfene Bibliothek ist das `DeviceListener` Interface der Einstiegpunkt, an dem Nutzer(innen) der Bibliothek ansetzen sollen, um ihre eigene Lösung zu verwirklichen.

Die zwei zu kapselnden Kernstücke, auf die sich die vorliegende Arbeit fokussiert - die serielle Verbindung und das Netzwerkgerät – können nun leicht in Projekte integriert werden.

### 4.3.6 AbstractProtocolLayer


- `connect()` – ruft die `connect()` Methode des Device Objektes auf und führt zusätzliche Prozesse aus, falls welche notwendig sind
- `disconnect()` – ruft die `disconnect()` Methode des Device Objektes auf und führt zusätzliche Prozesse aus, falls welche notwendig sind


### 4.3.7 ProtocolListener

Das `ProtocolListener` Interface fungiert ebenso wie das `SerialInputListener` Interface und das `DeviceListener` Interface als Verbindungsstück zur nächsten Abstraktionsebene. Nachdem Entwickler(innen) ein Protokoll entwickelt und als
4. Entwurf


Entwickler(innen) können und sollten das Interface entsprechend ihres Protokolls erweitern, in dem sie ein neues Interface implementieren, dass mit dem Schlüsselwort „extends“ von ProtocolListener erbt. Ein solches Interface könnte beispielsweise vorschreiben, dass seine Implementierungen Methoden wie onBroadcastReceived(), onResetReceived() oder onRouteRequestReceived() enthalten müssen. Sinnvoll ist das dann, wenn auf der Schicht über der Protocol Komponente Unterscheidungen notwendig sind; beispielsweise, wenn bestimmte Nachrichtentypen eine Veränderung der grafischen Oberfläche zur Folge haben sollen und andere nicht. Das ermöglicht Entwickler(innen) großen Spielraum, das Interface auf ihre Bedürfnisse zuzuschneiden. Dementsprechend sollte in der Subklasse von AbstractProtocolLayer ein Feld für den entsprechenden protokoll- spezifischen ProtocolListener vorhanden sein. Wäre dieser bereits in der abstrakten Oberklasse als Feld vom Typ ProtocolListener vorgegeben, so müssten Entwickler(innen) entweder umständliches Downcasting betreiben, oder das Feld verstecken und durch ein eigenes Feld ersetzen.
5 Implementierung

5.1 Aufspaltung in strukturbildende und beispielhafte Klassen


Im folgenden Abschnitt wird auf die Besonderheiten bei der Implementierung der Klassen eingegangen. Dabei werden in Abschnitt 5.2 die strukturbildenden Klassen diskutiert, während die Subklassen sowie die Erweiterung des ProtocolListener Interfaces in Abschnitt 5.3 behandelt werden.

5.2 Strukturbildende Klassen

5.2.1 AbstractSerialConnection und der SerialInputListener

Bei der Implementierung der Klasse AbstractSerialConnection wurde nach dem Klassendiagramm aus Abschnitt 4.3.1 vorgegangen. Zusätzlich wurden Setter- und Gettermethoden für das Attribut serialInputListener implementiert, sowie zwei Konstruktoren:

Ein Konstruktor der Form public AbstractSerialConnection(SerialInputListener serialInputListener), der dafür sorgt, dass der übergebene SerialInputListener das entsprechende Attribut initialisiert. Dieser Konstruktor ist dann sinnvoll, wenn Entwickler(innen) eine Klasse schreiben, die direkt auf eine SerialConnection zugreifen soll.

Zusätzlich gibt es einen parameterlosen Konstruktor, um es Subklassen zu ermöglichen, über den entsprechenden Setter auch nach der Instanzierung das Attribut SerialInputListener zu initialisieren. Das ermöglicht Flexibilität in der
Anwendung der Subklassen, hat jedoch den Nachteil, dass bei unachtsamer Instanzierung und Nutzung der Klassen eine NullPointerException auftreten kann. Für die Struktur der Bibliothek ist dieser parameterlose Konstruktor essenziell, da er es Subklassen ermöglicht, in ihrem Konstruktor super() ohne Parameter aufzurufen. Das nachträgliche Initialisieren des SerialInputListener kann so an die nächsthöhere Abstraktionsschicht delegiert werden, in diesem Fall an AbstractDevice.

Das SerialInputListener Interface kann ohne weitere Besonderheiten in Java implementiert werden, so wie es im Klassendiagramm aus Abbildung 7 dargestellt ist.

5.2.2 AbstractDevice und das DeviceListener Interface


```java
public AbstractDevice (AbstractSerialConnection serialConnection){
    serialConnection.setSerialInputListener(this);
    this.serialConnection = serialConnection;
}
```

Der zweite Konstruktor hat zusätzlich einen Parameter für ein DeviceListener Objekt, mit Hilfe dessen im Konstruktor der deviceListener gesetzt wird. Dieser Konstruktor ist für Situationen gedacht, in denen die entsprechende Subklasse von Entwickler(innen) direkt genutzt wird, ohne dass die Protocol Schicht darüber
gesetzt wird. Implementiert eine Android Activity Klasse das DeviceListener Interface, so kann sie den zweiten Konstruktor verwenden und sich selbst mit dem Schlüsselwort „this“ als DeviceListener an den Konstruktor übergeben.

Das Interface DeviceListener konnte - wie das SerialInputListener Interface - anhand des Klassendiagramms (Abbildung 7) ohne Besonderheiten umgesetzt werden.

5.2.3 AbstractProtocolLayer und das ProtocolListener Interface


```java
public AbstractProtocolLayer(AbstractDevice device) {
    device.setDeviceListener(this);
    this.device = device;
}
```

5. Implementierung

Entwickler(innen) können damit die im Rahmen dieser Arbeit anvisierte Konstruktorkette nutzen:

```java
ProtocolLayer protocol = new ProtocolLayer(this, new Device(new SerialConnection(targetAdress)));
```

Aufgrund der potenziell geringen Gemeinsamkeiten zwischen verschiedenen Protokollen wird dieser Konstruktor nicht in der abstrakten Oberklasse implementiert, sondern in die Hand der Entwickler(innen) gegeben, die diesen in Subklassen implementieren können.


Das ProtocolListener Interface wurde gemäß dem Klassendiagramm implementiert.

5.3 Implementierung der Subklassen und des ExampleProtocolListener Interface

5.3.1 Darstellung des Klassendiagramms mit Subklassen

5. Implementierung

5. Implementierung


5. Implementierung

5.3.3 Ting_01M

Das Hardware Modul Ting-01M ist das Funkmodul, welches dem Autor für diese Arbeit zur Verfügung gestellt wurde. Dementsprechend wurde die AbstractDevice Subklasse, die dieses Funkmodul abstrahieren soll, Ting_01M benannt. Das Ting-01M Modul kann über einen AT-Befehlssatz gesteuert werden, von dem jedoch nur eine begrenzte Teilmenge in der entsprechenden Java Klasse umgesetzt wurde, die dem Erreichen der Zielsetzung dieser Arbeit dienlich ist. Sämtliche mögliche Befehle sowie zusätzliche Dokumentation zur Kommunikation mit dem Hardwaremodul findet sich in chinesischer Sprache in [17]. Für die Implementation wurden die Funktionalitäten für das Senden von Nachrichten (AT+SEND), das Auslösen eines Resets (AT+RST) sowie das Einstellen der Gerätekonfiguration (AT+CONFIG) implementiert.

Die private `onMessageReceived()` Methode enthält die entsprechende Logik, wie mit den eingehenden Nachrichten zu verfahren ist.

Wird die `connect()` Methode der `Ting_01M` Klasse aufgerufen, dann erfolgt nach der Verbindung mit dem Gerät ein erster Reset und anschließend die Konfiguration des Gerätes mit Standardwerten. Die Methoden `configure()` und `configure(String)` erlauben es Entwickler(innen), andere Konfigurationen am Netzwerkgerät einzustellen. Die `reset()` Methode erlaubt einen Geräte Reset.

### 5.3.4 ExampleProtocolLayer und das ExampleProtocolListener Interface


In dem fiktiven Protokoll, dass der Klasse zugrunde liegt, gibt es drei Arten von Nachrichten: Reset-, Discovery und Broadcastnachrichten. Für die vorliegende Arbeit ist die Sinnhaftigkeit des Protokolls belanglos, die Trennung in diese drei Nachrichtentypen soll nur der Anschaulichkeit dienen.
5. Implementierung

Da diese Klasse lediglich als Strukturbispiel dienen soll, enthält sie keine tatsächliche Protokolllogik, sondern lediglich Teilimplementierungen. Dies wird am folgenden Codeausschnitt deutlich, der die Methode `onNetworkMessageReceived(String)` des `DeviceListener` Interfaces überschreibt:

```java
@Override
public void onNetworkMessageReceived(String message) {
    if (message.startsWith("DISCOVERY")) {
        protocolListener.onDiscoveryMessageReceived(message);
        // some code to answer to an incoming Discovery request
    } else if (message.startsWith("RESET")) {
        protocolListener.onResetMessageReceived(message);
        // some code to initiate a reset
    } else {
        protocolListener.onBroadcastMessageReceived(message);
        // some code to decide whether to forward the incoming broadcast to the neighbors or not
        // can be implemented here or in the class implementing ExampleProtocolListener
    }
}
```

Die Klasse `ExampleProtocolLayer` kümmert sich hauptsächlich darum, Nachrichten nach den entsprechenden Präfixen zu klassifizieren und an den registrierten `ExampleProtocolListener` weiterzuleiten. Zusätzlich bietet sie Funktionen, Nachrichten mit den entsprechenden Präfixen zu senden:

```java
public void sendBroadcast(String message) {
    device.send("BROADCAST,6TTL," + message);
}

public void sendNeighborDiscovery() {
    device.send("DISCOVERY,1TTL,Is anyone out there?");
}

public void sendNetworkReset() {
    device.send("RESET");
}
```

Die Implementierung des `ExampleProtocolListener` Interfaces birgt gegenüber seiner Darstellung im Klassendiagramm keine Besonderheiten und wurde wie dargestellt implementiert.
6 Test der entworfenen Lösung

6.1 Testaufbau und Testszenarien

Um die Funktionsfähigkeit der Bibliothek und ihre Struktur zu testen, soll für jede der drei in Abschnitt 4.1 und 4.2 beschriebenen Komponenten eine rudimentäre Android Applikation geschrieben werden. Die Applikationen binden jeweils die Bibliothek ein, setzen jedoch jeweils an einer anderen Komponente an. Dadurch sollen alle drei Abstraktionsstufen / Komponenten und ihr Zusammenspiel getestet werden. Hierbei stehen die funktionalen Anforderungen der Bibliothek im Vordergrund.

Für die Testszenarien steht der folgende Aufbau zur Verfügung:

- Ein Ting-01M Funkmodul das über einen UART-Adapter mit einem Bluetooth-Modul gekoppelt ist
- Ein Samsung Galaxy S10 mit der Android Version 9 auf dem die jeweilige Testapplikation installiert wird. Es soll sich per Bluetooth in jedem Testszenario mit dem obigen Modul verbinden
- Ein weiteres Ting01M Funkmodul, dass über ein USB TTL Kabel an einen USB OTG Adapter angeschlossen ist
- Ein Samsung Galaxy S7 mit der Android Version 8 und der Fremdapplikation Serial USB Terminal [18]. Es wird mit dem zuletzt genannten Funkmodul über den USB OTG Adapter verbunden
- Beide Ting-01M Module weisen die gleiche Grundkonfiguration auf und senden sämtliche Nachrichten als Broadcast an die für die Module bestimmte Broadcastadresse

In allen drei Testszenarien werden die Funktionen der Applikation durch eingehende und ausgehende Nachrichten getestet. Die Fremdapplikation Serial USB Terminal wird dabei als Kontrollinstrument und Kommunikationspartner verwendet. Über sie werden innerhalb der Testszenarien über AT-Befehle Nachrichten an das Funkmodul gesendet, das mit der Testapplikation verbunden ist. Gleichzeitig zeichnet es die von der Testapplikation an das Netzwerk
6. Test der entworfenen Lösung

ausgehenden Nachrichten auf. Abbildung 9 stellt diesen Testaufbau schematisch dar. Dabei zeigen die eingezeichneten blauen Pfeile die Kommunikationsflüsse.

![Diagramm der Testaufbau](image)

Abbildung 9: Schematischer Testaufbau

Jede der drei Testapplikationen besteht aus einer Android Activity Klasse „MainActivity“, die eines der drei Listener Interfaces implementiert (SerialInputListener, DeviceListener, ExampleProtocolListener). In den folgenden Abschnitten werden das Samsung Galaxy S10 und der damit verbundene Aufbau als „S10“ und das Samsung Galaxy S7 mit seinem Aufbau als „S7“ abgekürzt.

6.2 Testszenario 1: SerialInputListener

Das Ziel des ersten Testszenarios ist es, die SerialConnection Komponente und die dazugehörigen Klassen zu testen. Primär stehen hierbei die Überprüfung der Struktur sowie die Funktionsfähigkeit der BluetoothSerialConnection Klasse im Vordergrund. Sekundär geht es auch darum zu überprüfen, inwiefern Entwickler(innen) leicht Apps entwickeln können, die schon an dieser Stelle ansetzen.

Über die in Abbildung 10 dargestellte einfache grafische Benutzeroberfläche wird eine Eingabe von Text ermöglicht, der über einen Button an das Funkmodul gesendet werden kann. Ein Switch ermöglicht es, die `connect()` und `disconnect()` Funktionen der `BluetoothSerialConnection` aufzurufen. Zusätzlich zeigt ein Textfeld die ausgehenden und eingehenden Nachrichten an, die über die serielle Schnittstelle gesendet und empfangen werden.

Die `MainActivity` Klasse nutzt aus der für diese Arbeit entwickelten Bibliothek ein Feld `connection` vom Typen `BluetoothSerialConnection`, dass in der `onCreate()` Methode initialisiert wird:

```java
connection = new BluetoothSerialConnection(this,"98:D3:31:FB:3B:40");
```

Hier übergibt sich die `MainActivity` Klasse selbst als `serialInputListener` Parameter für den Konstruktor der `BluetoothSerialConnection`. 

**Abbildung 10: Grafische Nutzeroberfläche für Testszenario 1**

6. Test der entworfenen Lösung
6. Test der entworfenen Lösung

Anschließend wird das `connection` Objekt in den verschiedenen Listener Methoden der Benutzeroberflächenelemente verwendet, um bei Interaktionen mit diesen die entsprechenden Methoden `connect()`, `disconnect()` und `write(byte[] byte)` aufzurufen.

Schließlich muss die `MainActivity` Klasse noch die Methoden des `SerialInputListener` Interfaces implementieren. Beim Testen der entsprechenden Methoden stieß der Autor auf den folgenden Fehler:

```java
android.view.ViewRoot$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
```

Ausgelöst wird dieser davon, dass die `BluetoothSerialConnection` Klasse einen neuen Thread startet, um eingehende Nachrichten aus dem Eingangsbuffer der seriellen Schnittstelle zu lesen. Dieser neue Thread ruft `serialInputListener.onSerialDataReceived()` auf, innerhalb derer dafür gesorgt werden soll, dass sich der angezeigte Text auf der Benutzeroberfläche ändert. Der neue Thread hat jedoch hierzu keine Berechtigung; Elemente der Benutzeroberfläche dürfen nur von dem Thread verändert werden, der sie erstellt hat.


Schließlich sehen die vom `SerialInputListener` Interface überschriebenen Methoden wie folgt aus:

```java
@Override
public void onConnect() {
```
6. Test der entworfenen Lösung

```java
isConnected = true;
runOnUiThread(() -> {
    consoleText.setText("Successfully connected. \r\n");
});

@Override
public void onSerialDataReceived(byte[] bytes) {
    runOnUiThread(() -> {
        consoleText.append("[Serial]: " + bytes.toString() + "\r\n");
    });
}

@Override
public void onError( Exception e) {
    runOnUiThread(() -> {
        consoleText.append("[ERROR]: " + e.toString() + "\r\n");
    });
}

Die Variable `consoleText` steht hierbei für das Textelement der grafischen Oberfläche, auf der die ein- und ausgehenden Nachrichten protokolliert werden.

Für das Testszenario werden zusätzlich zum erfolgreichen Verbinden der Testapplikation mit dem Funkmodul folgende Fälle getestet:

1. S10 sendet einen String mit dem Inhalt „AT“
2. S10 sendet eine Nachricht mit einem beliebigen String als Payload, der nicht mit AT beginnt
4. S7 sendet eine Nachricht an das Netzwerk über den Befehl „AT+SEND=5“ und nach der Bestätigung des Funkmoduls den Payload „hello“

Im Fall 1 wird erwartet, dass das Netzwerkgerät mit „AT,OK“ antwortet und diese Nachricht auf der Benutzeroberfläche erscheint.

Im Fall 2 wird erwartet, dass das Netzwerkgerät einen Fehler zurückmeldet und „AT,ERR:CMD“ auf der Konsole von S10 erscheint.

Im Fall 3 wird erwartet, dass S7 eine Nachricht erhält, die den korrekten Payload von S10 enthält.
6. Test der entworfenen Lösung

Im Fall 4 wird erwartet, dass S10 eine Nachricht enthält, die mit „LR,“ beginnt und den korrekten Payload von S7 enthält. Die Nachricht wird auf der Konsole von S10 angezeigt.

6.3 Testszenario 2: DeviceListener


Für dieses Testszenario wurde ebenfalls ein neues Android Projekt in Android Studio erstellt, das eine einzige Activity Klasse „MainActivity“ enthält. Diese Klasse implementiert das *DeviceListener* Interface und hat aus der für diese Arbeit entwickelten Bibliothek ein Feld *device* vom Typen *Ting_01M*, dass in der *onCreate()* Methode initialisiert wird:

```java
device = new Ting_01M(this, new BluetoothSerialConnection("98:D3:31:FB:3B:40"));
```

Abbildung 11: Grafische Nutzeroberfläche für Testszenario 2

Abbildung 11 stellt die grafische Benutzeroberfläche der Testapplikation dar. Entsprechend der Implementation der Klassen der Device-Komponente gibt es hier
6. Test der entworfenen Lösung

zwei Buttons zum Senden von Nachrichten an die serielle Schnittstelle, anders als im ersten Testszenario: Einer zum Senden von Nachrichten an das Netzwerk, der die send(String) Methode von device aufruft. Und einen, der die sendDeviceCommand(String) Methode aufruft.

Analog zu Testszenario 1 sehen die vom DeviceListener Interface überschriebenen Methoden wie folgt aus:

```java
@Override
public void onConnect() {
    isConnected = true;
    runOnUiThread(() -> consoleText.setText("Successfully connected. \r\n"));
}

@Override
public void onDeviceMessageReceived(String message) {
    runOnUiThread(() -> consoleText.append("[Device] " + message + "\r\n"));
}

@Override
public void onNetworkMessageReceived(String message) {
    runOnUiThread(() -> consoleText.append("[Network]: " + message + "\r\n"));
}

@Override
public void onError(Exception e) {
    runOnUiThread(() -> consoleText.append("[ERROR]: " + e.toString() + "\r\n"));
}
```

Auch sie dienen ausschließlich zur Ausgabe der empfangenen Nachrichten.

Für das Testszenario werden zusätzlich zum erfolgreichen Verbinden mit dem Funkmodul folgende Fälle getestet:

1. S10 sendet einen Befehl an das Netzwerkgerät mit dem Inhalt „AT“
2. S10 sendet eine Nachricht mit einem beliebigen String als Payload an das Netzwerk
3. S7 sendet eine Nachricht mit einem beliebigen String als Payload an das Netzwerk
6. Test der entworfenen Lösung

Im Fall 1 wird erwartet, dass das Netzwerkgerät mit „AT,OK“ antwortet und diese Nachricht auf der Benutzeroberfläche erscheint. Sie muss außerdem korrekt als Nachricht vom Gerät klassifiziert werden.

Im Fall 2 wird erwartet, dass die Nachricht bei S7 mit dem korrekten Payload ankommt und auf der Konsole in der Serial USB Terminal App erscheint.

Im Fall 3 wird erwartet, dass die entsprechenden Nachrichten korrekt als Nachricht aus dem Netzwerk klassifiziert und auf der Nutzeroberfläche angezeigt werden.

6.4 Testszenario 3: ExampleProtocolListener


```java
```

Die entsprechenden Methoden der Klasse können nun verwendet werden; durch connect() wird die seriele Verbindung zum Netzwerkgerät aufgebaut.

Abbildung 12 zeigt die Nutzeroberfläche für das dritte Testszenario:
6. Test der entworfenen Lösung

Abbildung 12: Grafische Nutzeroberfläche für Testszenario 3

Die Nutzeroberfläche ähnelt den beiden anderen Testszenarien. Die Klasse `ExampleProtocolLayer` stellt drei Funktionen zum Senden von Nachrichten an das Netzwerk zur Verfügung, für die jeweils ein Button auf der Benutzeroberfläche erstellt wurde. Dadurch wird die Funktion des zweiten Testszenarios erweitert. Der Button für das Senden von AT-Befehlen an das Netzwerkgerät bleibt hier erhalten, jedoch ändert sich der Aufruf im Code:

```java
protocolNode.getDevice().sendDeviceCommand(message);
```

Über die `getDevice()` Methode des `ExampleLayerProtocol` Objektes kann auf das `Ting_01M` Objekt und seine Methode `sendDeviceCommand(String)` zugegriffen werden. Selbstverständlich sind die einzugebenden Befehle hier stark von dem benutzten Netzwerkgerät abhängig.

Die restlichen drei Buttons werden mit den Methoden von `ExampleProtocolLayer` verknüpft: `sendBroadcast(String)`, `sendReset()` und `sendNetworkDiscovery()`.

Die obigen Funktionen sind unabhängig von der darunterliegenden Device Ebene. Entwickler(innen) müssen kein Wissen über das zugrundeliegende Gerät haben, um diese Methoden sinnvoll in ihre Applikation zu implementieren.
6. Test der entworfenen Lösung

Die vom `ExampleProtocolListener` Interface überschriebenen Methoden sind wie folgt gestaltet:

```java
@override
public void onConnect() {
    isConnected = true;
    runOnUiThread(() -> consoleText.setText("Successfully connected. \r\n"));
}

@override
public void onDeviceMessageReceived(String message) {
    runOnUiThread(() -> consoleText.append("[Device] " + message + "\r\n"));
}

@override
public void onBroadcastMessageReceived(String message) {
    runOnUiThread(() -> consoleText.append("[Network Broadcast]: " + message + "\r\n"));
}

@override
public void onDiscoveryMessageReceived(String message) {
    // in this example, there is no need to process the parameter
    // "message"
    runOnUiThread(() -> consoleText.append("[Network]: Received a Neighbor Discovery request." + "\r\n"));
}

@override
public void onResetMessageReceived(String message) {
    // in this example, there is no need to process the parameter
    // "message"
    runOnUiThread(() -> consoleText.append("[Network]: Received a reset message." + "\r\n"));
}

@override
public void onError(Exception e) {
    runOnUiThread(() -> consoleText.append("[ERROR]: " + e.toString() + "\r\n"));
}
```


Für das Testszenario werden zusätzlich zum erfolgreichen Verbinden mit dem Funkmodul folgende Fälle getestet:

1. S10 sendet einen Befehl an das Netzwerkgerät mit dem Inhalt „AT“
6. Test der entworfenen Lösung

2. S10 sendet an S7 eine Nachricht mit dem Wort „RESET“ am Anfang des Payload, wenn der „Network Reset“ Button gedrückt wird
3. S10 sendet an S7 eine Nachricht mit dem Wort „DISCOVERY“ am Anfang des Payload, wenn der „Neighbor Discovery“ Button gedrückt wird
4. S10 sendet an S7 eine Nachricht mit einem beliebigen anderen String als Payload über den „Send Broadcast“ Button
5. S7 sendet an S10 eine Nachricht mit dem Payload „RESET“
6. S7 sendet an S10 eine Nachricht mit dem Payload „DISCOVERY“
7. S7 sendet an S10 eine Nachricht mit einem beliebigen anderen String als Payload

Im Fall 1 wird erwartet, dass das Netzwerkgerät mit „AT,OK“ antwortet und diese Nachricht auf der Benutzeroberfläche erscheint. Sie wird außerdem korrekt als eine Nachricht vom Gerät klassifiziert.

In den Fällen 2 bis 4 wird erwartet, dass die Nachricht bei S7 mit dem korrekten Payload ankommt und auf der Konsole in der Serial USB Terminal App erscheint.

In den Fällen 5 bis 7 wird erwartet, dass die entsprechenden Nachrichten korrekt als Netzwerknachrichten und zusätzlich als Reset, Neighbor Discovery oder Broadcast Nachricht eingestuft und auf der Nutzeroberfläche angezeigt werden.
7 Ergebnisbewertung der Testszenarien

In diesem Abschnitt sollen die Ergebnisse der vorangestellten Testszenarien zusammengefasst werden. Alle drei Testszenarien konnten mithilfe der Bibliothek und der in ihr enthaltenen Klassen erfolgreich implementiert werden.


7. Ergebnisbewertung


Durch die Implementierung der drei Subklassen BluetoothSerialConnection, Ting_01M und ExampleProtocolLayer sowie des Interfaces ExampleProtocolListener wurde ebenfalls implizit die Erweiterbarkeit der Bibliothek getestet, da sie die vorgegebenen Strukturen nutzen.

Ein für die Softwareentwicklung wesentlicher Aspekt, die Performance, wurde in den dargestellten Testszenarios nicht berücksichtigt.
8 Zusammenfassung

Das Ziel dieser Arbeit, eine Bibliothek zu entwerfen, die in neue Androidprojekte eingebunden werden kann um schnell und leicht die Kommunikation über ein drahtloses ad-hoc Netzwerk aufzubauen, wurde erfüllt. Es konnte eine geeignete Struktur aufgebaut werden, anhand derer Applikationen für den Aufbau von ad-hoc Multihopnetzwerken mit wenigen Zeilen Code programmiert werden können.


Die erfolgreich durchgeführten Testszenarien belegen eine grundlegende Funktionsfähigkeit der Bibliothek und ihrer Komponenten in einem Netzwerk mit zwei Netzwerkteilnehmern. Sie weisen darauf hin, dass die Bibliothek und ihre Struktur dafür geeignet sind, die Kommunikation für drahtlose ad-hoc Netzwerke zu unterstützen.

Über die Nutzung von abstrakten Klassen und Interfaces gibt die innerhalb dieser Arbeit entwickelte Lösung ein Grundmaß an Struktur vor, um die in ihr enthaltenen Komponenten sinnvoll einzusetzen. Dadurch unterstützt sie die Arbeit von Entwickler(innen), in deren Projekte sie eingebunden wird. Die spezifischen Eigenschaften der seriellen Verbindung über Bluetooth und des Ting-01M Netzwerkgerätes sind durch die entsprechenden Klassen gekapselt, sodass Entwickler(innen) sich wenig mit ihnen beschäftigen müssen.
9 Ausblick

Die im Rahmen dieser Arbeit entstandene Lösung hat eine Möglichkeit aufgezeigt, wie die grundlegenden Kommunikationsflüsse sinnvoll abstrahiert werden können, um so eine leichte Entwicklung von Androidapplikationen für ad-hoc Netzwerke zu ermöglichen. Aufgrund des explorativen Charakters dieser Arbeit wurden mehrere Aspekte vernachlässigt, die in weiterführenden Arbeiten von Relevanz sein könnten.


Der zweite Aspekt ist die Performanz, die für die vorliegende Arbeit nicht berücksichtigt wurde. Die Implementierung der BluetoothSerialConnection als eigener Thread, der die serielle Schnittstelle regelmäßig abtastet, könnte möglicherweise nachteilig gegenüber anderen Lösungen sein - wie beispielsweise dem aktiven Abrufen aus dem Hauptthread. Da für Netzwerkapplikationen die Performanz ein nicht unwesentlicher Faktor sein kann, wären weitere Untersuchungen hier sinnvoll.
9. Ausblick

Des Weiteren könnte die Struktur der Bibliothek ausgiebigeren Tests unterzogen werden. Sowohl die Nutzbarkeit als auch die Funktionen wurden in einem eng gesteckten Rahmen geprüft. Eventuelle Schwierigkeiten, die bei der Entwicklung einer umfangreicherer Applikation auftreten oder sich durch eine größere Anzahl an Netzwerkteilnehmern ergeben könnten, können schwer antizipiert werden.
Literaturverzeichnis


Anhang

A.1. Inhalt der beiliegenden SD Card

- Verzeichnis Bachelorarbeit PDF:
  Die vorliegende Arbeit in digitaler Version als PDF-Dokument
- Verzeichnis Quellen:
  Alle Online-Quellen, welche im Literaturverzeichnis aufgeführt sind
- Verzeichnis Quellcode:
  Der Quellcode der entwickelten Bibliothek
- Verzeichnis Testszenarien:
  Der Quellcode der Testszenarien

A.2. Link zum Code Repository auf Github.com

https://github.com/HWittlich/AdHocNetworkCommunication
Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, habe ich kenntlich gemacht.
Ebenso versichere ich, dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Berlin, 26.08.2019 _________________________

Hagen Wittlich