HomeAid

Heimautomatisierung für Pflegebedürftige

Matthias Böffel
Jonas Collet
Tobias Habermann
Markus Nebel

Fachhochschule Kaiserslautern
University of Applied Sciences

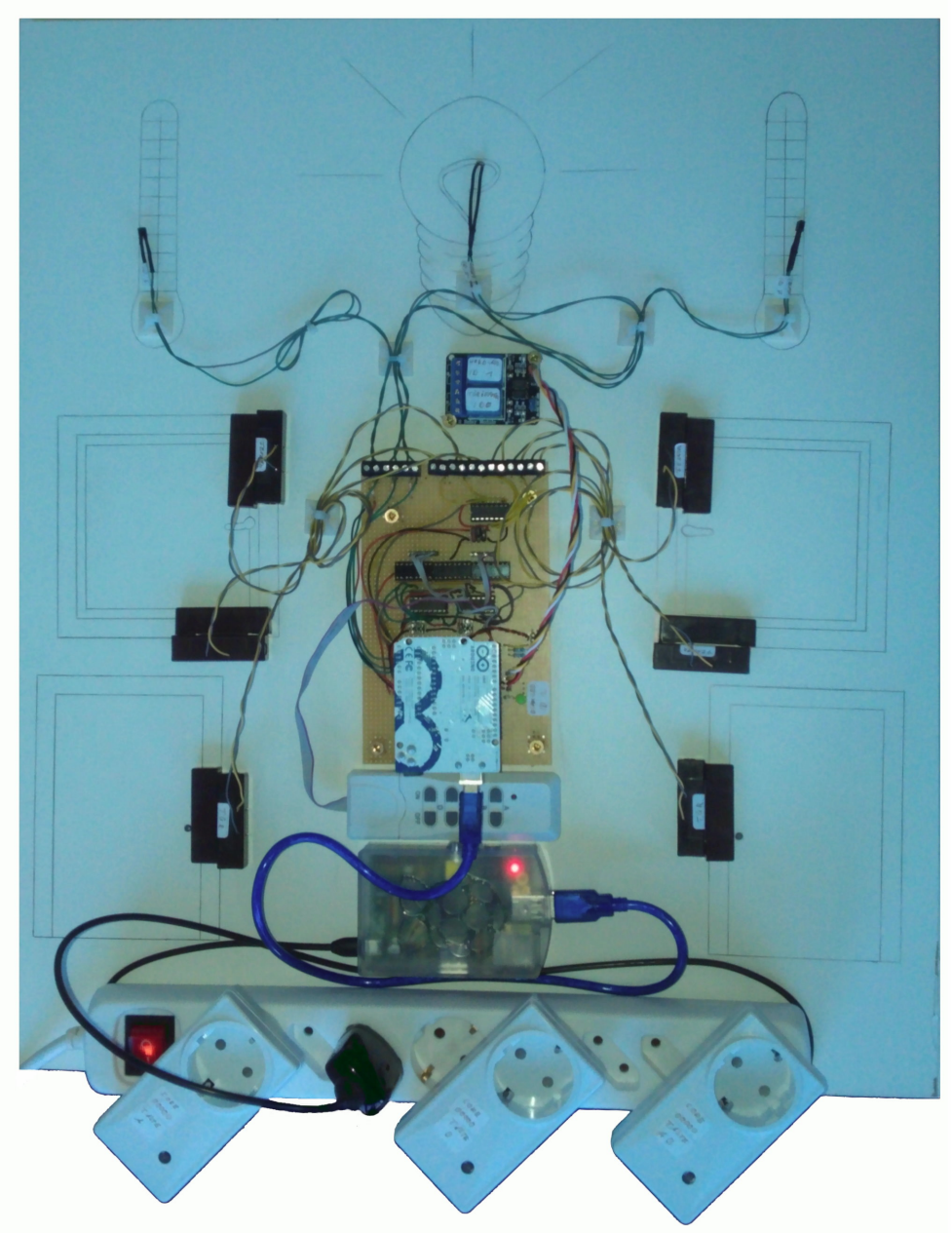
29. Juni 2014

Studien-Projekt
Betreuer: Dr.-Ing. Hubert Zitt

Fachhochschule Kaiserslautern | University of Applied Sciences

Abstract

Im fünften Semester des Studiengangs „Angewandte Informatik“ soll im Rahmen des sogenannten Studienprojekts und innerhalb einer Gruppe eine praktische Projektarbeit umgesetzt werden. Innerhalb dieses Studienprojekts wurde unter der Betreuung von Dr. Hubert Zitt ein Heimautomatisierungssystem entwickelt das pflegebedürftige Menschen unterstützen und deren Betreuung erleichtern soll.



Inhaltsverzeichnis

1 Einleitung

Unter kontinuierlicher Abstimmung der Projektparameter bearbeitet jedes Teammitglied einen eigenen Teilbereich dieses Projekts und das zugehörige Kapitel innerhalb dieser Dokumentation. Zwischen den Komponenten wurden zusätzlich Vorgaben zur Kommunikation festgelegt um eine gute Erweiterbarkeit und Austauschbarkeit zu gewährleisten.

Die zentrale Komponente ist hierbei der Server, auch als *HomeAid Control System* bezeichnet. Er abstrahiert die spezifischen Kommunikationsprotokolle unterschiedlichster Hardware, und stellt die in ein universelles Format gebrachten Daten der Android-App und dem Webinterface zur Verfügung. Der Server wurde von Markus Nebel mit einem Zeitaufwand von ca. 205 Stunden implementiert.

Die verwendete Hardware wurde als Prototyp ausgearbeitet und steht stellvertretend für jede Form von Heimautomatisierungshardware. Der Prototyp wurde von Tobias Habermann entworfen und konstruiert. Es wurde ein Zeitaufwand von 145 Stunden investiert.

Die Android-App soll sowohl den Pflegebedürftigen als auch das Pflegepersonal bzw. andere betreuende Parteien benutzerfreundlich unterstützen. Während die Steuerung und Überwachung der betreffenden Wohnung eher für Tablet-Systeme angepasst wurde, sind die Alarmierungsfunktionen für alle Android-Endgeräte gleich nützlich. Die Android-App wurde von Matthias Böffel implementiert. Eine Einarbeitung in die Thematik der Android-Programmierung war notwendig und es war insgesamt ein Zeitaufwand von 213 Stunden zur Umsetzung erforderlich.

Die Weboberfläche realisiert zusätzlich eine plattformunabhängige Möglichkeit, die räumlichen Gegebenheiten über eine einfache Oberfläche einzusehen. Die Weboberfläche wurde von Jonas Collet implementiert. Dafür waren ca. 130 Stunden Arbeit notwendig.

In der folgenden Dokumentation werden die einzelnen Komponenten der Projektarbeit detailliert beschrieben.

2 Middleware (Raspberry Pi)

Die Middleware stellt die Zentrale des HomeAid Systems dar. Sie vermittelt zwischen der Hardware und bietet höheren Anwendungen, wie der Android App, eine Schnittstelle um die von ihm aufbereiteten Daten abzufragen.

Das **HomeAid Control System** ist vollständig in C++ nach dem C++11 Standard implementiert und auf Unix-basierende Systeme zugeschnitten. Entwickelt wurde die Anwendung auf einem Linux-System. Unter anderem wurde sie auf dem stromsparenden Raspberry Pi auf Funktion getestet.

2.1 Komponenten

In Abbildung 1 werden die einzelnen Software-Komponenten und deren Zusammenhänge im HomeAid Control System gezeigt.

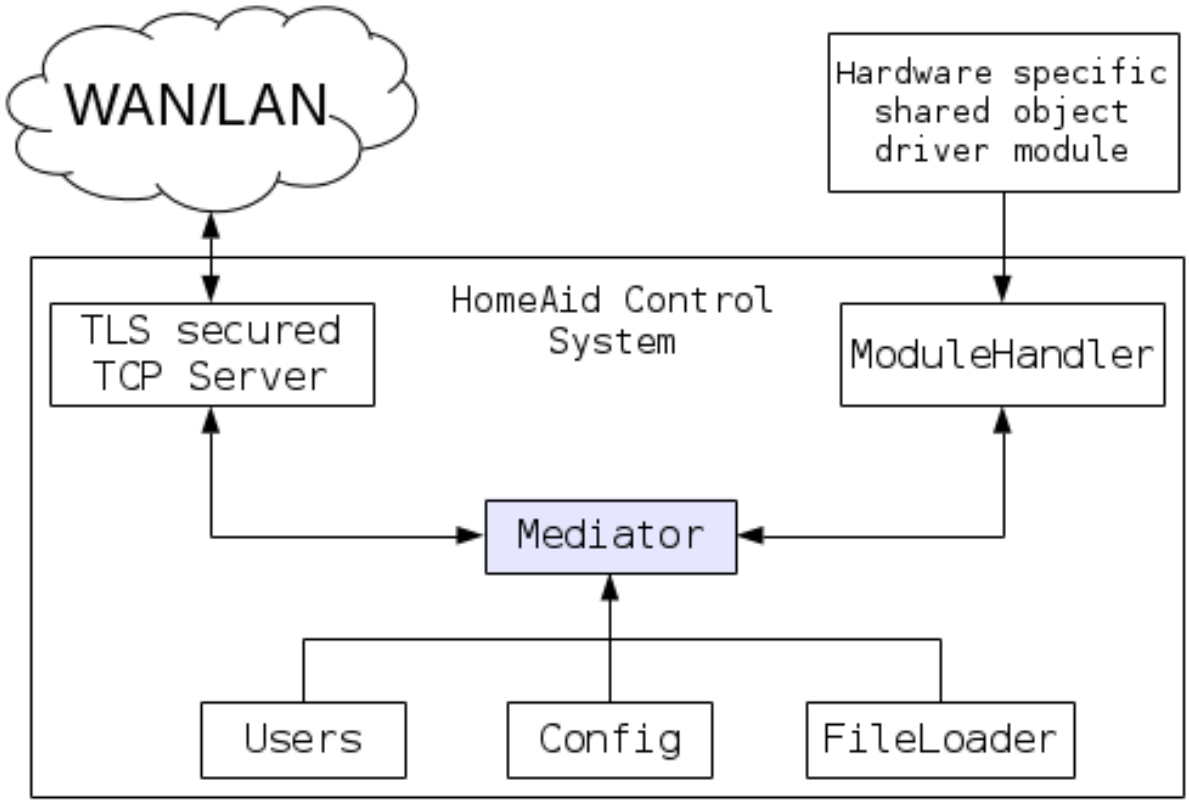


Abbildung 1: Programmaufbau

Im Folgenden werden die Komponenten und deren Funktion genauer erläutert. Die Funktionsweise der Shared Object Modules wird im Kapitel 2.2 detailliert erklärt.

2.1.1 Mediator

Der Mediator (Vermittler) bildet den Kern der HomeAid Control Anwendung. In Abbildung 2 wird der interne Aufbau visualisiert. Der Mediator vermittelt die Daten zwischen den verschiedenen Schnittstellen und verwaltet die Anmeldungen der Benutzer über den TCP Server, sowie die Abstraktion der verfügbaren Sensoren. Das verwendete Protokoll

für die Socket-Kommunikation wird ebenso im Mediator bearbeitet, wie das Triggern von Alarm-Signalen an die über das Netzwerk verbundenen Benutzer.

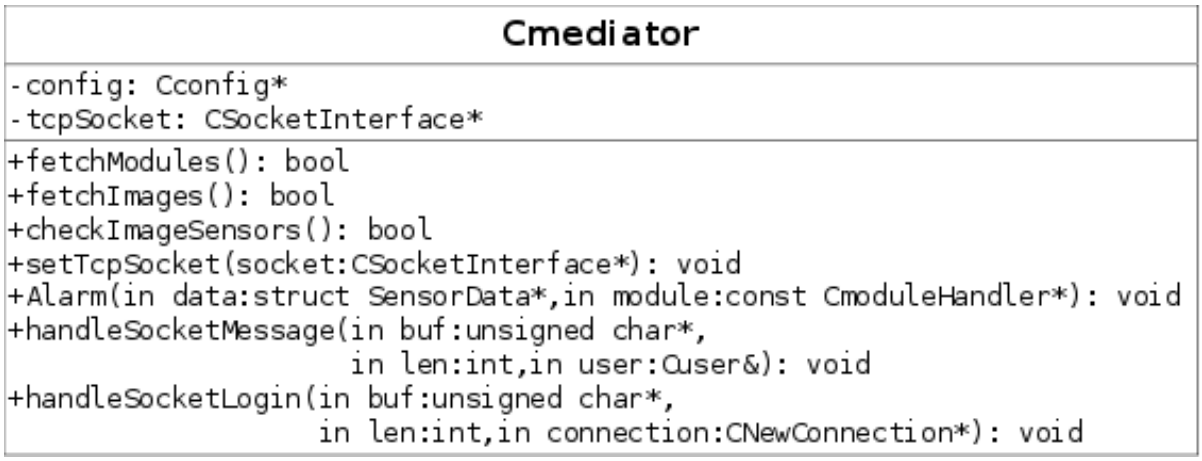


Abbildung 2: Klassendiagramm Medaitor

2.1.2 FileLoader

Sämtliche Bild-Dateien, welche die Räume repräsentieren, werden ebenfalls vom HomeAid Control System verwaltet und über die Socketschnittstelle zur Verfügung gestellt. Um die Dateien beim Start der Anwendung in den Speicher zu laden, startet der Mediator für jede in der Konfiguration definierte Bild-Datei den *FileLoader*, welcher die Datei einliest und seinem entsprechenden Raum zuordnet.

2.1.3 Konfiguration

Das HomeAid Control System wird über eine XML-Datei konfiguriert. Die Konfiguration wird von der Klasse *Cconfig* verwaltet. Sie liest die Konfigurationsdatei ein, parst deren XML Baumstruktur und legt die Konfigurations-Werte in ihren Attributen ab. Die *Cconfig*-Klasse ist mit dem Singleton-Pattern implementiert, um nur eine Instanz von ihr erzeugen zu können. Über die *config.xml* können folgende Eigenschaften konfiguriert werden:

- Pfade zu Shared-Object-Module-Files
- Benutzerdaten für die Anmeldung über die TCP-Schnittstelle
- Räume
- Bilder mit Zuordnung zu diesen Räumen
- Sensoren mit Zuordnung zu ihrem Modul und Position auf dem Bild

Die in Listing 1 dargestellte Beispiel-Konfiguration für das HomeAid Control System zeigt eine minimale Konfiguration mit zwei Modulen, zwei Benutzern die über den TCP Server verbinden können, einem Raum und einem Bild für diesen Raum mit zwei konfigurierten Sensoren.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<config>
  <!-- define a list of shared object modules to load on startup -->
  <modules>
    <module>
       <!-- relative path to shared object: -->
      <file>modules/exampleModule1.so</file>
      <!-- identifier for this module -->
      <id>0</id>
    </module>
    <module>
      <file>modules/exampleModule2.so</file>
      <id>1</id>
    </module>
  </modules>
  <!-- define a list of user accounts -->
  <users>
    <user>
      <username>TestUser1</username>
      <password>testP455W0rd1</password>
    </user>
    <user>
      <username>TestUser2</username>
      <password>testP455W0rd2</password>
    </user>
  </users>
  <!-- define a list of rooms -->
  <rooms>
    <room>
      <name>TestRoom1</name>
      <!-- unique room identifier: -->
      <id>0</id>
      <!-- list of images for this room, identified by their
         image identifier -->
      <images>
        <id>0</id>
      </images>
    </room>
  </rooms>
  <!-- define a list of images with their sensor information -->
  <images>
    <image>
      <!-- specify the path to the image file -->
      <file>sample_apartment/guest_room_and_floor.png</file>
      <!-- give it a name -->
      <name>Guestroom and Floor</name>
      <!-- identifier of this image -->
      <id>0</id>
      <!-- list of sensors positionized on this image -->
      <sensors>
        <sensor>
          <!-- id of the module which handles this sensor -->
          <module>1</module>
          <!-- id of this sensor IN the module, see config of module -->
          <id>1</id>
          <!-- door sensor -->
          <type>1</type>
          <!-- x and y coordinates: -->
          <x>330</x>
          <y>50</y>
        </sensor>
        <sensor>
          <module>0</module>
          <id>1</id>
          <type>3</type>
          <x>750</x>
          <y>250</y>
        </sensor>
      </sensors>
    </image>
  </images>
</config>
```

Listing 1: Beispiel-Konfiguration

2.1.4 TCP Server

Der TCP Server stellt eine TLS gesicherte IPv4 Netzwerk-Schnittstelle zur Verfügung. Der Server nimmt neue Verbindungen an und leitet über den Mediator einen per Timeout-gesicherten Login-Vorgang ein. Alle eingehenden Nachrichten werden an einen Message-Handler im Mediator übergeben, der deren Gültigkeit überprüft und über ein Command-Interface entsprechend reagiert. Der Timeout beim Login stellt sicher, dass ein sich anmeldender Client nicht dauerhaft die TCP Ressourcen blockieren kann. Außer dem `LOGIN`-Kommando steht einem **noch nicht** authentifizierten Client keine weiteren Kommandos zur Verfügung.

2.1.5 Benutzer

Für die Authentifizierung über die TCP Server Schnittstelle werden die Benutzerinformationen in der Klasse *Cuser* abgelegt und beim Login verglichen. Ein Benutzer kann nur mit einem Gerät zur gleichen Zeit eingeloggt sein. Weitere Login-Versuche werden abgelehnt.

2.1.6 ModuleHandler

Der ModuleHandler stellt die Verwaltung optionaler Hardware-Schnittstellen zur Verfügung. In Abbildung 3 ist der Klassenaufbau des ModuleHandlers dargestellt. Er bindet Shared Object Module Code zur Laufzeit ein, und lädt deren exportierten Funktionszeiger ins Hauptprogramm. Über diese Funktionszeiger kann das Modul initialisiert und zerstört werden. Der ModuleHandler nimmt über die Initialisierungs-Funktion ein Klassen-Objekt des *ModuleInterface*-Basistypen vom Shared Object entgegen. Durch dynamische Polymorphie kann über diese Schnittstelle die komplette Kommunikation mit dem konkreten Modul-Objekt im Shared Object realisiert werden.

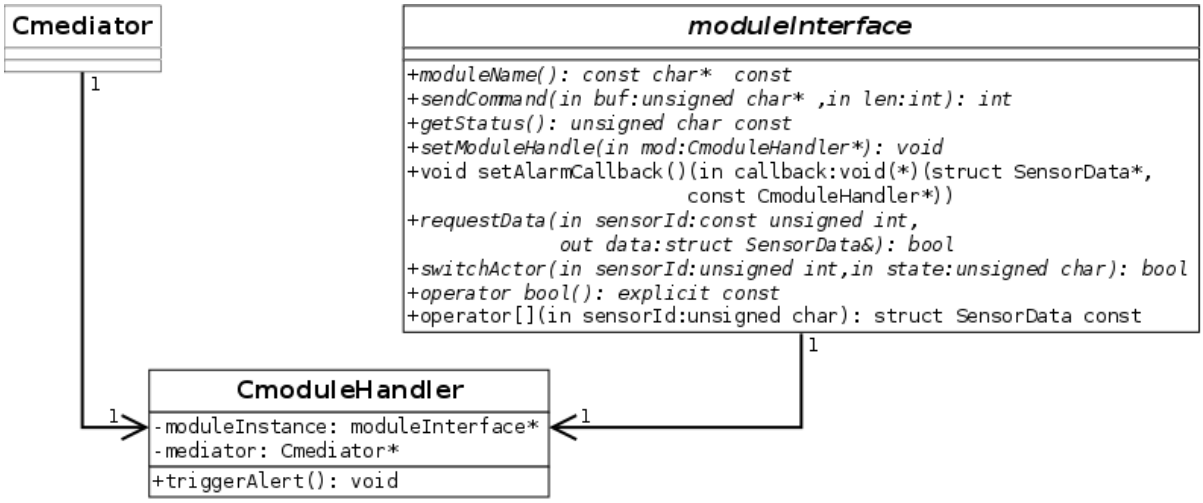


Abbildung 3: Klassendiagramm ModuleHandler

2.2 Shared Object Modules

Um das Verwenden verschiedenster Hardware zu ermöglichen, nutzt die HomeAid Control Anwendung zur Laufzeit ladbare Shared Object Modules (unter Linux mit der Dateiendung *.so*). Die Module enthalten den für die jeweilige Hardware benötigte spezifischen

Code um an die Informationen der eingesetzten Sensoren zu gelangen und liefern diese anschließend über ein definiertes Klassen-Interface an das Hauptprogramm. Dadurch wird der hardwarespezifische Teil der Anwendung in externe Module ausgelagert, wodurch keine Anpassungen am Hauptprogramm mehr notwendig sind, wenn neue Hardware eingesetzt wird. Lediglich ein damit kompatibles Modul muss verfügbar sein, ähnlich der Treiber-Modulen die von einem Betriebssystem-Kern nachgeladen werden können, um neue Peripherie anzusteuern. Jedes Shared Object kann seine eigene Konfigurationsdatei verwenden um modulspezifische Einstellungsmöglichkeiten nutzen zu können.

Das Shared Object muss die zwei exportierten Funktionen *initModule()* und *destroyModule()* bereitstellen. Mit *initModule()* muss ein Objekt einer Klasse, die die *ModuleInterface*-Basisklasse implementiert, erzeugt, und ein Zeiger des Typs *ModuleInterface* zurückgeliefert werden.

Abbildung 4 zeigt den schematischen Ablauf dieser Module-Laderoutine durch Abarbeitung folgender Schritte:

- Öffnen des Shared Object Files
- Laden der Function-Pointer zu den exportierten Funktionen
- Aufruf der exportierten *initModule()*-Funktion, welche das Module-Object erzeugt
- Erhalt eines Pointers auf dieses Objekt als *ModuleInterface*-Typ
- Zum Ende der Lebenszeit des ModuleHandlers: Aufruf der *destroyModule()*-Funktion

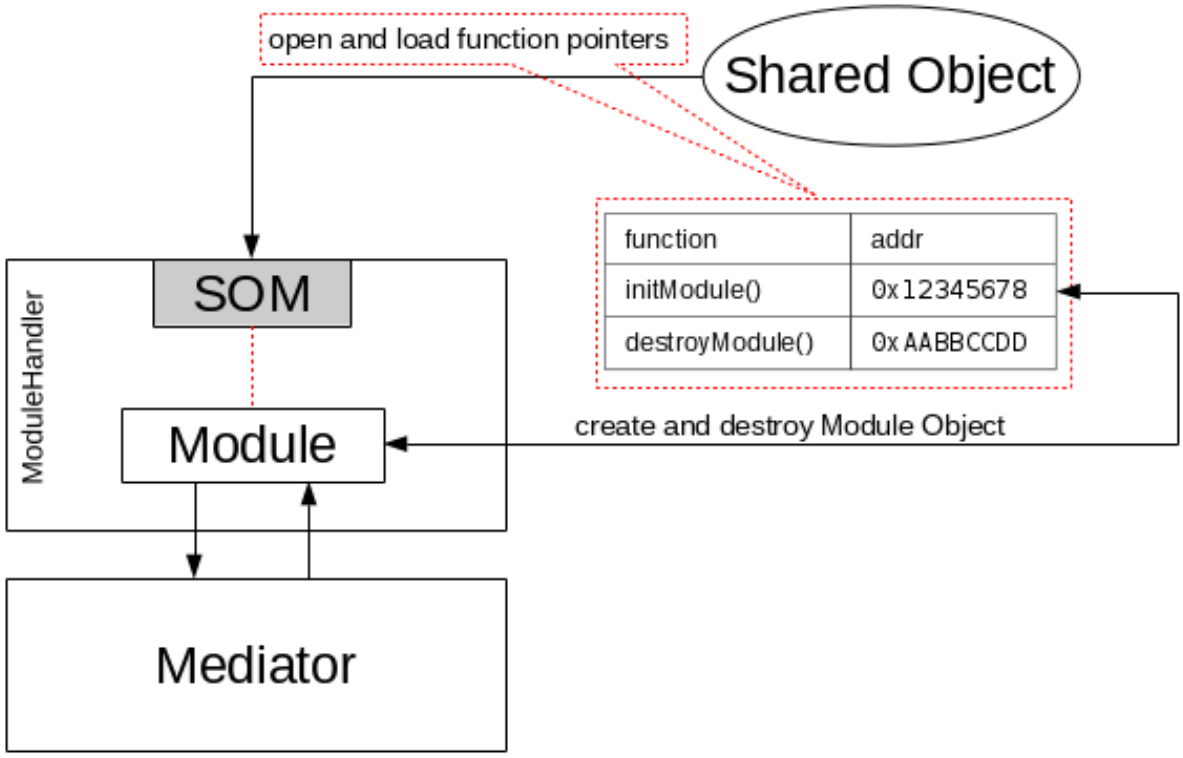


Abbildung 4: Laden des Shared Object Codes, Erzeugen des Modul Objekts

2.3 Schnittstellen-Protokoll

Zur Kommunikation mit externen Anwendungen wurde ein eigenes Protokoll entwickelt. Dieses wird im Folgenden genauer erklärt.

2.3.1 Protokoll-Header

Für alle Pakete gilt der folgende Header:

Feld	Bit-Breite	Beschreibung
CMD	8	Kommando
MsgLen	16	Anzahl der folgenden Bytes

Es folgt der Payload mit einer Länge von *MsgLen*-Bytes, mit ggf. zusätzlichen Daten die vom Kommando erwartet werden. Ist *MsgLen* gleich 0, enthält die Nachricht keinen Payload.

2.3.2 KEEPALIVE

Das `KEEPALIVE`-Kommando dient dazu, von Seiten einer externen Anwendung die TCP Verbindung am Leben zu halten. Es sollte mindestens alle 300 Sekunden einmal gesendet werden. Wird es längere Zeit nicht gesendet, kann der Server die Verbindung zur Sicherheit schließen.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
Random-Pad	$8 * MsgLen$	Zufallzahlen

Das Random-Pad wird 1:1 in die Antwort kopiert und kann eine Länge von 5 bis 25 Byte haben. Es ist vom Client frei wählbar.

2.3.3 LOGIN

Das `LOGIN`-Kommando nutzt der Client um sich mit Benutzername und Passwort gegenüber dem Server zu authentifizieren und eine Benutzersitzung zu initiieren.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
UsernameLen	16	Länge des Benutzernamens
Username	$8 * UsernameLen$	ASCII-Benutzername (ohne $\backslash 0$)
PasswordLen	16	Länge des Passworts
Password	$8 * PasswordLen$	ASCII-Klartext-Passwort (ohne $\backslash 0$)

Die maximal erlaubte Länge für Benutzername und Passwort beträgt 99 Zeichen.

Bei fehlerhaftem Login wird die Verbindung geschlossen. Bei Erfolg wird folgende Antwort gesendet:

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
Payload	16	0x00 0x01

2.3.4 GET_ROOMS

Mit dem `GET_ROOMS`-Kommando kann der Client alle zur Verfügung stehenden Räume mit deren Bezeichnung abfragen.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1, kein Payload

Die Antwort enthält neben der Anzahl der Räume, je nach *NumberOfRooms*, eine oder mehrere *RoomDescriptions*:

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
NumberOfRooms	16	Anzahl der Räume
RoomDescriptor	$Var * NumberOfRooms$	Raum Beschreibung

Aufbau einer *RoomDescription*:

Feld	Bit-Breite	Beschreibung
RoomId	16	Room Identifier
NameLen	16	Länge des Namens
Name	$8 * NameLen$	ASCII-Name (ohne $\backslash 0$)
NumberOfImages	16	Anzahl der Bilder für diesen Raum
ImageDescriptor	$144 * NumberOfImages$	Enthält die ImageIds und Prüfsummen

Es folgen *NumberOfImages* Image Descriptors:

Feld	Bit-Breite	Beschreibung
ImageIds	$16 * NumberOfImages$	Liste von Image Idenfiers
MD5	$128 * NumberOfImages$	MD5 Prüfsumme für jedes Bild

2.3.5 GET_IMAGE

Mit dem `GET_IMAGE`-Kommando kann der Client ein Bild einer bestimmten ImageId abfragen. Das Bild wird im HomeAid Control System aus einer Datei in den Speicher gelesen und binär zum Client geschickt.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
ImageId	16	Image Identifier

Die Antwort weicht in diesem Fall leicht vom restlichen Standard ab. Es wird keine Länge übermittelt, da die Bilddateien weitaus größer sein können als das Längen-Feld im Header Bytes angeben kann.

Feld	Bit-Breite	Beschreibung
CMD	8	Kommando
ImageId	16	Image Identifier
Image	$8 * Var$	Das Bild

2.3.6 GET_IMAGE_INFO

Mit dem `GET_IMAGE_INFO`-Kommando kann der Client die Sensor-Informationen zusammen mit den Koordinaten auf dem zugehörigen Bild abfragen.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
ImageId	16	Image Identifier

Die Antwort enthält, je nach Anzahl der konfigurierten Sensoren, einen oder mehrere *SensorDescriptors*.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
ImageId	16	Image Identifier
NumberOfSensors	16	Anzahl der folgenden SensorDescriptions
SensorDescriptor	$NumberOfSensors * 136$	Eine oder mehrere SensorDescriptions

Der Aufbau eines *SensorDescriptor*:

Feld	Bit-Breite	Beschreibung
ModuleId	16	Identifier des Moduls, welches den Sensor besitzt
SensorId	32	Identifier des Sensors
Type	16	Typ des Sensors
X	16	X-Koordinate auf dem Bild
Y	16	Y-Koordinate auf dem Bild
StateFlags	8	Verfügbarkeitsinformation des Sensors
SensorState	32	Status des Sensors (je nach Type)

2.3.7 SWITCH_ACTOR

Mit dem `SWITCH_ACTOR`-Kommando kann der Client den Zustand eines bestimmten Aktors umschalten.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
ImageId	16	Image Identifier
ModuleId	16	Module Identifier
SensorId	32	Sensor Identifier
SwitchAction	8	Zielzustand des Aktors

Bei Erfolg wird die Antwort zum `GET_IMAGE_INFO`-Kommando gesendet (siehe 2.3.6).

2.3.8 ALERT

Das `ALERT`-Kommando wird vom Server ohne vorherigen Request an alle Clients mit zu diesem Zeitpunkt bestehender TCP Verbindung gesendet. Es soll höheren Anwendungen die Möglichkeit bieten ihre Anwender durch akustische oder visuelle Signale über einen

kritischen Zustand zu informieren. Zusätzlich kann für jeden Sensor ein Alarm-Text definiert werden, der ebenfalls mit dieser Nachricht verschickt wird.

Feld	Bit-Breite	Beschreibung
Header	24	Siehe 2.3.1
ModuleId	16	Module Identifier
ImageId	16	Image Identifier
SensorId	32	Sensor Identifier
Type	16	Sensor Typ
TextLen	16	Länge des Alarm-Textes
AlarmText	$8 * TextLen$	ASCII-Text (ohne $\backslash 0$)

2.4 Externe Libraries

Zum Parsen der XML Konfigurationsdateien wird die *RapidXml* C++ Bibliothek [1] verwendet. Sie wird beim Kompiliervorgang statisch ins HomeAid Control Binary gebunden und benötigt somit keine zusätzliche Installation der vorkompilierten Bibliothek.

2.5 Metriken

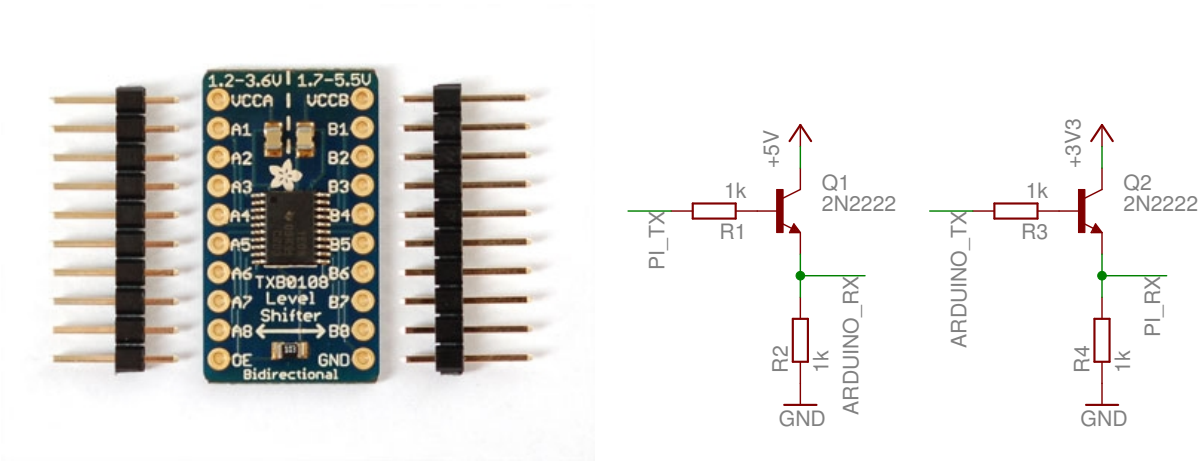
Anzahl der Programmcodezeilen insgesamt: 6745
Anzahl der Programmcodezeilen ohne XML-Lib: 4050

3 Hardware (Arduino)

Im Rahmen des Projekts wurde eine eigene Hardwarekomponente (HomeAid-Hardwaresystem) entwickelt. Diese verfügt über verschiedene Sensoren, die von einem Arduino Uno verwaltet werden. Er ist über eine serielle Schnittstelle mit dem Control-Server, auf dem das Serverprogramm läuft, verbunden und gibt die Daten bei einer Änderung oder auf Anfrage weiter. Für dieses Projekt wurde der Control-Server auf einem RaspberryPi installiert. Des Weiteren nimmt der Arduino alle Kommandos für die Aktoren entgegen und führt die entsprechenden Aktionen aus.

3.1 Der Arduino

Die Kommunikation zwischen dem Arduino und dem Control-Server kann entweder per USB oder bei einem RaspberryPi direkt über dessen GPIO-Pins erfolgen. Die Kommunikation per USB ist mit den entsprechenden Treibern ohne weiteres möglich. Möchte man die Kommunikation mit Hilfe der GPIO-Pins des RaspberryPi realisieren, muss man jedoch beachten, dass der Arduino mit einer Spannung von $5V$ arbeitet, der RaspberryPi jedoch nur mit $3,3V$. Die simpelste Lösung ist die Anpassung der Spannung mittels eines Pegelwandlers (Abbildung 5(a)). Die einfachere und günstigere Variante ist jedoch, selbst eine Pegelanpassung mit Hilfe einer einfachen Transistorschaltung (Abbildung 5(b)) vorzunehmen. In beiden Fällen muss beachtet werden, dass RX und TX der beiden Komponenten über Kreuz, also TX des RaspberryPi an RX des Arduino und TX des Arduino an RX des RaspberryPi, verbunden werden müssen.



(a) 8-Kanal Pegelwandler [2]

(b) Transistorschaltung

Abbildung 5: Pegelwandler und Transistorschaltung

3.2 Schaltplan

In Abbildung 6 wird der gesamte Schaltplan des HomeAid-Hardwaresystems gezeigt. In der nachfolgenden Tabelle werden die wichtigsten Bauteile benannt.

Bauteil	Bezeichnung	Bemerkung
IC1 - IC3	PCF8574P	8-Bit I^2C I/O-Expander
OK1A - OK3D	OKY74-4H	4 Kanal Optokoppler
JP1		Tastencode der Fernbedienung
JP2		Systemcode der Fernbedienung
JP3		Stromversorgung der Fernbedienung
S1	Reedschalter	Türsensor 1
S2	Reedschalter	Fenstersensor 1
S3	Reedschalter	
S4	Reedschalter	Türsensor 2
S5	Reedschalter	Fenstersensor 2
S6	Reedschalter	
PH1	A 9050 14	Lichtsensor
THERMO1	KTY 81-220	Temperatursensor 1
THERMO2	KTY 81-220	Temperatursensor 2

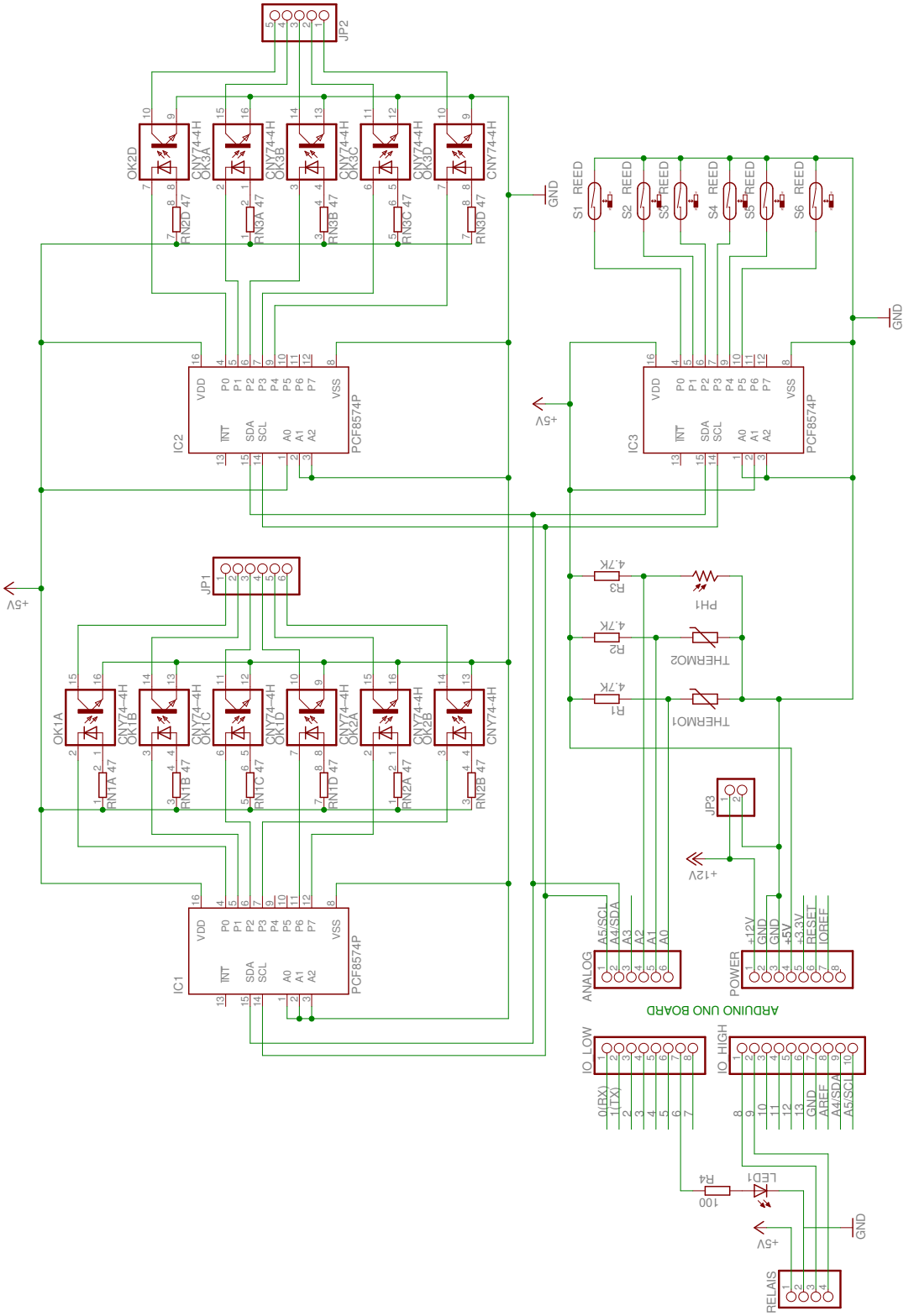


Abbildung 6: Schaltplan

3.3 Steckdosen und Fernbedienung

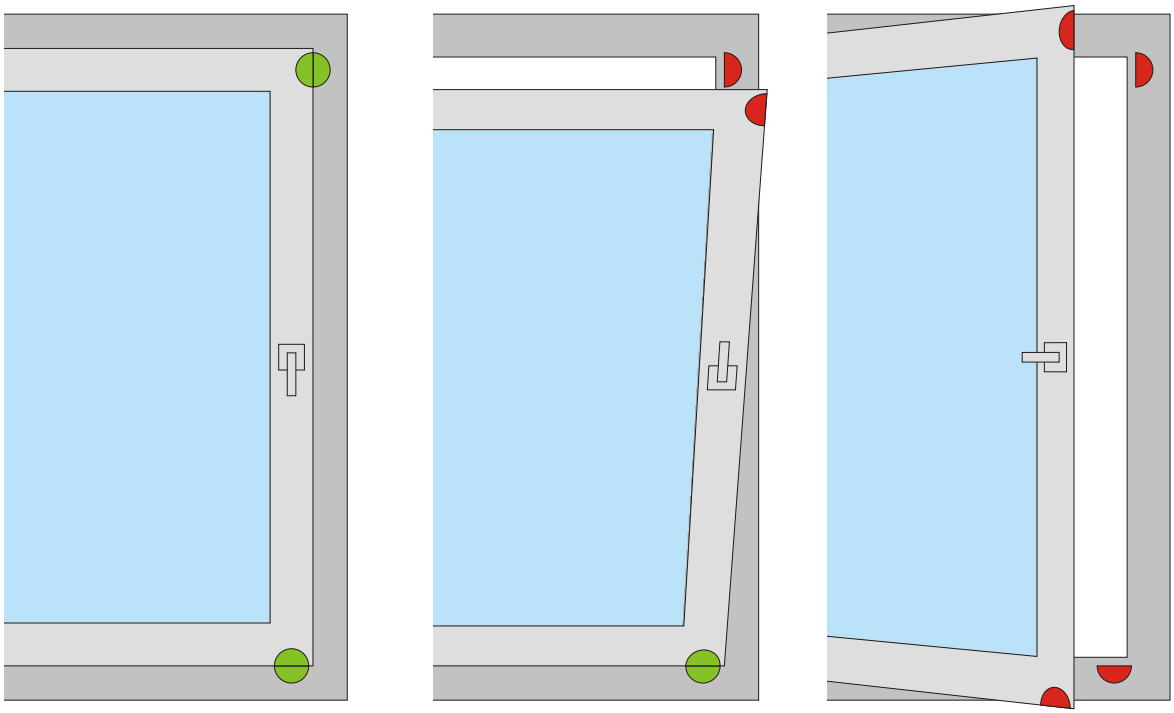
Um elektrische Endverbraucher ein- und ausschalten zu können, wurde ein Funksteckdosenset in das HomeAid-Hardwaresystem mit eingebunden. Hierbei wurde mit Hilfe von Optokopplern die Fernbedienung des Funksteckdosensets mit dem Arduino verbunden. Die Zwischensteckdosen selbst wurden hierbei nicht modifiziert, so dass keine VDE-Richtlinien verletzt wurden oder neu geprüft werden mussten.
Mit einem Optokoppler lassen sich Schaltkreise galvanisch trennen. Der Optokoppler enthält pro Kanal eine Leuchtdiode (aktiver Teil) und einen Fototransistor (passiver Teil), die in einem undurchsichtigen Gehäuse untergebracht sind. Sobald die Leuchtdiode leuchtet, schaltet der Fototransistor durch.
Zum Umschalten der Steckdosen werden bei der Fernbedienung jeweils 2 Pins des IC's der Fernbedienung mit GND verbunden (siehe Tabelle). Verbindet man nun die entsprechenden Pins mit dem passiven Teil der Optokoppler und deren aktiven Teil mit dem Arduino, kann dieser das „Drücken" der Tasten übernehmen[3]. Drückt man auf der Fernbedienung mehrere Tasten - beispielsweise A und B - gleichzeitig, werden nicht alle Steckdosen mit dem Tastencode A und dem Tastencode B geschaltet, sondern nur die mit dem Tastencode AB. Somit lassen sich eine wesentlich größere Anzahl von Steckdosen als auf den ersten Blick erwartet steuern. Das Drücken von mehreren Tasten ist zwar von Hand sehr umständlich, aber für den Arduino ohne Probleme bequem möglich.

Befehl	Pin 6	Pin 7	Pin 8	Pin 10	Pin 12	Pin 13
A ein	X				X	
A aus	X					X
B ein		X			X	
B aus		X				X
C ein			X		X	
C aus			X			X
D ein				X	X	
D aus				X		X

Der Systemcode, der dazu dient, mehrere Steckdosensysteme mit verschiedenen Fernbedienungen gleichzeitig unabhängig voneinander nutzen zu können, wird mittels eines DIP-Schalters codiert. Auch hier werden wieder einzelne Pins auf das Massepotential verbunden. Daher kann der DIP-Schalter durch weitere Optokoppler ersetzt und so der Systemcode dynamisch eingestellt werden.

3.4 Tür- und Fenstersensoren

Als Tür- und Fenstersensoren wurden bei diesem System Reedschalter verwendet. Reedschalter reagieren auf ein Magnetfeld und können je nach Bauweise, wenn ein Magnetfeld vorhanden ist, ein-, aus- oder umschalten. Sobald dieses Magnetfeld nicht mehr vorhanden ist, schalten sie wieder zurück in den Ursprungszustand. Zur Nutzung als Türsensor wird ein Reedschalter am Rahmen nahe der Tür angebracht. Ein Dauermagnet wird so an der Tür montiert, dass dieser, wenn die Tür geschlossen ist, den Reedschalter schließt. So kann durch das Auslesen des Reedschalters festgestellt werden, ob die Tür geöffnet oder geschlossen ist. Wie in Abbildung 7 gezeigt, werden an einem Fenster zwei Reedschalter angebracht: Einer unten und der andere an der sich öffnenden Seite. Damit kann man unterscheiden, ob das Fenster komplett geöffnet oder nur gekippt ist.



(a) geschlossen (b) gekippt (c) geöffnet

Abbildung 7: Zustände der Sensoren beim Fester

3.5 Temperatursensor

Zur Messung der Temperatur wurde ein Silizium-Temperatursensor eingesetzt. Dieser verändert seinen Leitwert abhängig von der Temperatur; je niedriger die Temperatur desto geringer der Widerstand. Um nun mit dem Arduino etwas messen zu können, wird ein hochohmiger Festwiderstand in Reihe geschaltet, so dass die beiden Bauteile einen Spannungsteiler bilden. Diese Spannung lässt sich mit dem Arduino messen und in eine Temperatur umrechnen. Aus den Werten des Datenblatts lässt sich anhand der Temperatur und dem zugehörigen Widerstandswert erkennen, dass das Verhältnis fast linear ist. Das nachfolgende Diagramm (Abbildung 8) zeigt die Kennlinie des Temperatursensors (blau) und den theoretischen Verlauf einer linearen Kennlinie (rot) zwischen den beiden Randwerten.

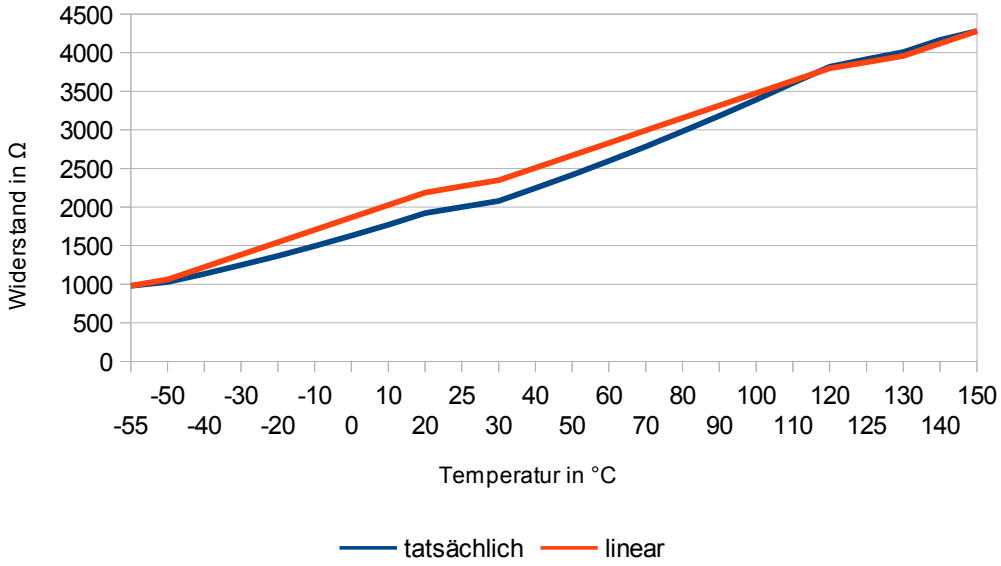


Abbildung 8: Kennlinie des Temperatursensors

Auf Grund der Tatsache, dass der hier verwendete Temperatursensor eine relativ hohe Ungenauigkeit aufweist (durchschnittlich $\pm 5°C$ Abweichung gegenüber den Standardwerten), wurde auf eine Berechnung der Temperatur anhand der Standardwerte des Datenblatts verzichtet. Stattdessen wurde mit einem anderen Temperaturmessgerät eine möglichst niedrige und eine möglichst hohe Temperatur gemessen und die jeweils vom Arduino festgestellten Werte als Referenz notiert. Mithilfe der folgenden Formel, die bereits in der `map()`-Funktion des Arduinos implementiert ist, lassen sich alle Messwerte in Temperaturen umrechnen. Hierbei steht $Temp$ für die Temperaturreferenzwerte und Ref für die jeweils vom Arduino gemessenen Werte.

$$Temperatur = (gemessener\ Wert - Ref_{low}) \cdot \frac{\Delta Temp}{\Delta Ref} + Temp_{low}$$

3.6 Lichtsensor

Als Lichtsensor kam ein Fotowiderstand zum Einsatz. Er verhält sich ähnlich wie der Temperatursensor: Mit steigender Lichtintensität verringert sich sein Widerstand. Auch hier wird durch eine Reihenschaltung mit einem Festwiderstand ein Spannungsteiler gebildet. Da im Datenblatt keine exakten Daten über den Widerstandswert bei konkreten Lichtintensitätswerten vorhanden sind, sondern man diese nur aus einer schlecht dargestellten Kennlinie entnehmen kann, wurde entschieden, den vom Arduino gemessenen Wert nicht umzurechnen. Ein genauer Lichtintensitätswert ist in diesem Fall auch nicht unbedingt erforderlich, da der Lichtsensor nur dazu dient, „Licht an“ und „Licht aus“, also hell und dunkel unterscheiden zu können.

3.7 Türöffner

Das HomeAid-Hardwaresystem verfügt über die Möglichkeit, einen Türöffner zu steuern. Hierbei wird ein einfaches Relais parallel zu dem Türöffnertaster in den Schaltkreis des Türöffners an der Haustür eingesetzt. Da der Arduino nicht genug Strom zur Verfügung stellt, um das Relais direkt zu schalten, wird ein Transistor zur Verstärkung benötigt. Des Weiteren wird eine Schutzdiode benötigt, da der selbstinduzierte Strom den Arduino beim Ausschalten des Relais sonst zerstören würde. In Abbildung 9 wird gezeigt, wie eine solche Schaltung aussehen könnte. Der Einfachheit halber wurde hier ein fertiger Relaisbaustein verwendet. Zusätzlich zu der eben beschriebenen Schaltung besitzt dieser einen Optokoppler zur galvanischen Trennung vom Steuersystem.

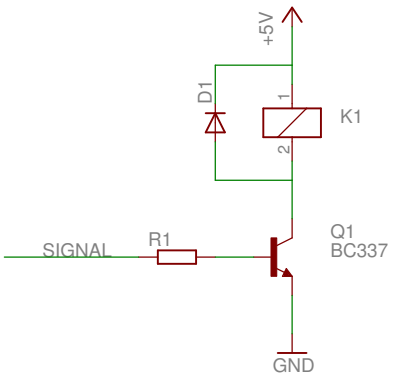


Abbildung 9: Einfache Relaisansteuerung mit Schutzdiode

3.8 Notausschalter

Der Notausschalter ist nur beispielhaft im HomeAid-Hardwaresystem realisiert. Er soll später dazu dienen, elektrische Verbraucher, die möglicherweise nicht über eine Steckdose angeschlossen sind (wie beispielsweise ein Herd), bei Bedarf abschalten zu können. Stellvertretend ist hier ein Relais eingebaut, das mit einer inversen Logik angesteuert wird. Das heißt: Solange das Steuersignal logisch Null ist, schließt das Relais den Stromkreis des Verbrauchers. Somit ist es im Ausgangszustand eingeschaltet und muss nicht erst vom Benutzer aktiviert werden.

3.9 PWM-Dimmer

Bei der Pulsweitenmodulation (PWM) wird bei konstanter Periodendauer die Pulsdauer eines Rechtecksignals verändert. In Verbindung mit einer hohen Frequenz kann das menschliche Auge die einzelnen Impulse nicht mehr differenzieren. Eine mit einem PWM-Signal betriebene LED wird gedimmt wahrgenommen. Da in den Haushalten immer mehr LED-Leuch-mittel zum Einsatz kommen, die sich mit einem herkömmlichen Dimmer nicht dimmen lassen, werden zukünftig immer mehr PWM-Dimmer benötigt. Die dimmbare LED des HomeAid-Hardwaresystems stellt beispielhaft eine LED-Zimmerbeleuchtung dar, die mithilfe des HomeAid-Systems gedimmt werden kann.

3.10 Software

Die Software auf dem Arduino dient dazu, die Werte der Sensoren zu ermitteln und gesammelt an den Control-Server weiterzugeben. Zu Beginn kann der Control-Server eine Liste der Sensoren und Aktoren anfordern. Die im Arduino gespeicherte Liste enthält Informationen, an welchem Pin der jeweilige Sensor oder Aktor angeschlossen ist, ob er direkt am Arduino oder an einem I^2C-Baustein angeschlossen ist, ob eine inverse Logik (beispielsweise bei einem Relais, das eingeschaltet wird, sobald es mit GND verbunden wird) anzuwenden ist und den zuletzt gemessenen Wert. An den Control-Server werden aus dieser Liste nur Typ und ID der Sensoren und Aktoren übermittelt, da alle anderen Informationen nur für den Arduino relevant sind.
Im nächsten Schritt kann der Control-Server alle Sensorwerte abfragen und die *AutoSend*-Funktion aktivieren. In diesem Modus vergleicht der Arduino die aktuellen Messwerte mit den gespeicherten Messwerten des letzten Durchgangs und sendet eine Meldung, sobald sich einer von ihnen geändert hat. Das zyklische Überprüfen der Sensoren wird mittels eines Timerinterrupts angestoßen.
Der Control-Server kann jederzeit Kommandos zum Umschalten eines Aktors senden. Die Kommandos werden vom Arduino zunächst im seriellen Buffer gespeichert und dann in der Arduino-Funktion `serialEvent()` abgearbeitet. Diese wird immer dann aufgerufen, wenn ein Durchlauf der Arduino-Funktion `loop()` beendet ist und Daten im seriellen Buffer verfügbar sind. Sie wird also hier immer aufgerufen, sobald ein Befehl ankommt und der Arduino nicht gerade mit der Ausführung eines anderen Befehls beschäftigt ist, da die `loop()`-Funktion leer ist.
Der Arduino kann zum Betätigen der Aktoren vier verschiedene Befehle entgegennehmen. Der erste Befehl dient zum einfachen Ein- oder Ausschalten eines Aktors. Ein weiteres Kommando ermöglicht es, einen Aktor für eine bestimmte Zeit ein- oder auszuschalten. Nach Ablauf dieser Zeit wird der Aktor zurück in seinen ursprünglichen Zustand geschaltet. Damit kann beispielsweise ein Türöffner betätigt werden. Die beiden anderen Befehle ermöglichen es, die PWM-Kanäle des Arduino zu nutzen. Mit ihnen kann ein Aktor, der an einem PWM-Kanal angeschlossen ist, stufenweise angesteuert werden. Auch hier gibt es einen Befehl, um dauerhaft einen neuen Wert einzustellen und einen, um nach Ablauf einer bestimmten Zeit zum vorherigen Wert zurückzuschalten.
Des Weiteren ist zum Umschalten der Steckdosen eine Art Transaktionskonzept implementiert. Da 512 verschiedene Steckdosen angesteuert werden können (5 Bit Systemcode, 4 beliebig kombinierbare Tasten; $2^5 * 2^4 = 512$) und der Arduino Uno 1 KB EEPROM (nichtflüchtiger, wieder beschreibbarer Speicher) zur Verfügung stellt, kann jede Steckdose problemlos mindestens ein Byte zur Zustandsspeicherung verwenden. Da der EEPROM des Arduino byteadressierbar ist und den Steckdosen mit Hilfe von Systemcode und Tastenkombination jeweils eine eindeutige ID zugeordnet werden kann, kann man ohne weitere Rechenoperationen jeder Steckdose einen Speicherplatz zuordnen. Dazu wird die ID der Steckdose als Speicheradresse des Status dieser Steckdose verwendet. Zur Realisierung des Transaktionskonzepts werden nur zwei Bits benötigt. Ein Bit speichert den aktuellen Zustand, in dem anderen Bit wird vor dem tatsächlichen Umschalten der Sollzustand gespeichert. Solange das Umschalten komplett durchgeführt werden kann, beinhalten beide Bits immer denselben Wert, da der tatsächliche Zustand nach dem Umschalten aktualisiert wird. Sollte ein Umschaltvorgang nicht komplett ausgeführt werden können (z.B. Verlust der Betriebsspannung während dem Umschalten), kann der Arduino den Umschaltvorgang beim Starten fortsetzen, wenn er feststellt, dass in einem Byte die beiden Bits nicht dieselben Werte haben.

4 Android-App

4.1 Funktionalität

Die App für Android-Betriebssysteme ab Version 4.1 (Jelly Bean) ist zur Verwendung sowohl durch den Pflegebedürftigen als auch durch Pflegepersonal geeignet. Für den Pflegebedürftigen bietet die App ständige Kontrolle über Sensoren und Schalter im Haushalt. So kann er jederzeit überprüfen, ob und wo im Haushalt Fenster und Türen geöffnet sind, Stromquellen eingeschaltet sind und welche Temperatur in den Räumen vorherrscht. Das Öffnen der Haustür und Betätigen von Schaltern bzw. Steckdosen kann der Pflegebedürftige ebenfalls durchführen, während das Pflegepersonal von der Alarmfunktion profitiert, welche in HomeAidControl konfigurierte Ereignisse, wie z.B. zu hohe Temperatur oder gefährliche Pulswerte durch einen Alarmton meldet. HomeAidControl bildet die sogenannte „Middleware“ zwischen der beliebigen Hardware und den Bediensoftware-Elementen und wird im folgenden Control-Server genannt. In der App können mehrere Haushalte eingetragen und überwacht werden und es werden die deutsche und englische Sprache gleichermaßen unterstützt. Im Folgenden wird der Aufbau der Applikation durch die Kapitel *Frontend* und *Backend* unterschieden.

4.2 Frontend

Mit Frontend ist an dieser Stelle das User-Interface und dessen Aufbau bezeichnet (Abbildung 10). Daher werden in den folgenden Unterkapiteln eher der jeweilige Aufbau der Bildschirmseite und die Eingabemöglichkeiten beschrieben, da sich die Verarbeitungsschritte auf Code-Ebene hier nicht signifikant unterscheiden. Die Meisten Bildschirmseiten behandeln Formulareingaben und das Reagieren auf Auswahl und Touch-Events. Eine Ausnahme bildet die komplexere Klasse *ImageViewFragment*, deren Ablauf im zugehörigen Unterkapitel etwas genauer beschrieben wird.

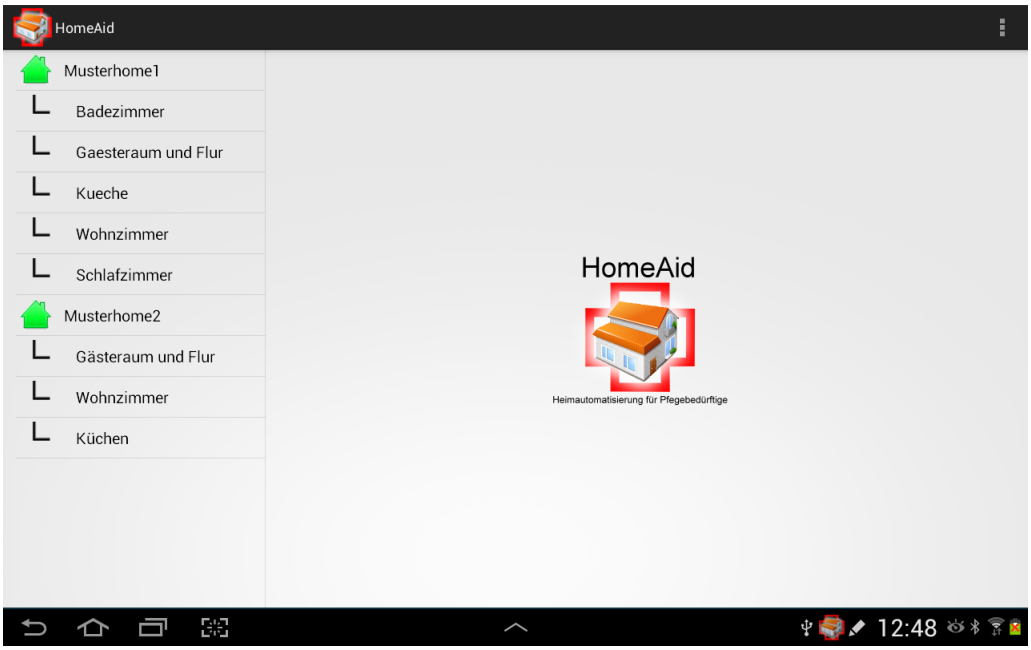


Abbildung 10: Screenshot Oberfläche

4.2.1 HomeAidActivity

Durch die Android-Tablet optimierte Programmierung mittels Fragments wurde hier eine „Eltern“-Activity angelegt, die sich um die Anzeige und Verwaltung der Fragments kümmert. Fragments können die Bildschirmseite in mehrere Bereiche aufteilen. In jedem dieser Bereiche kann dann eine eigene Fragmentseite angezeigt werden. In dieser Klasse *HomeAidActivity* wird je nach Display-Größe entschieden, welches Anzeigeformat verwendet werden soll: Zwei Fragments pro Seite (Menüleiste und Inhalt) bei großen Geräten (siehe Abbildung 11) und ein Fragment pro Seite wenn die Bildschirmseite nicht genügend Platz bietet (siehe Abbildung 12). Die *HomeAidActivity* wird beim Aufruf der App gestartet.

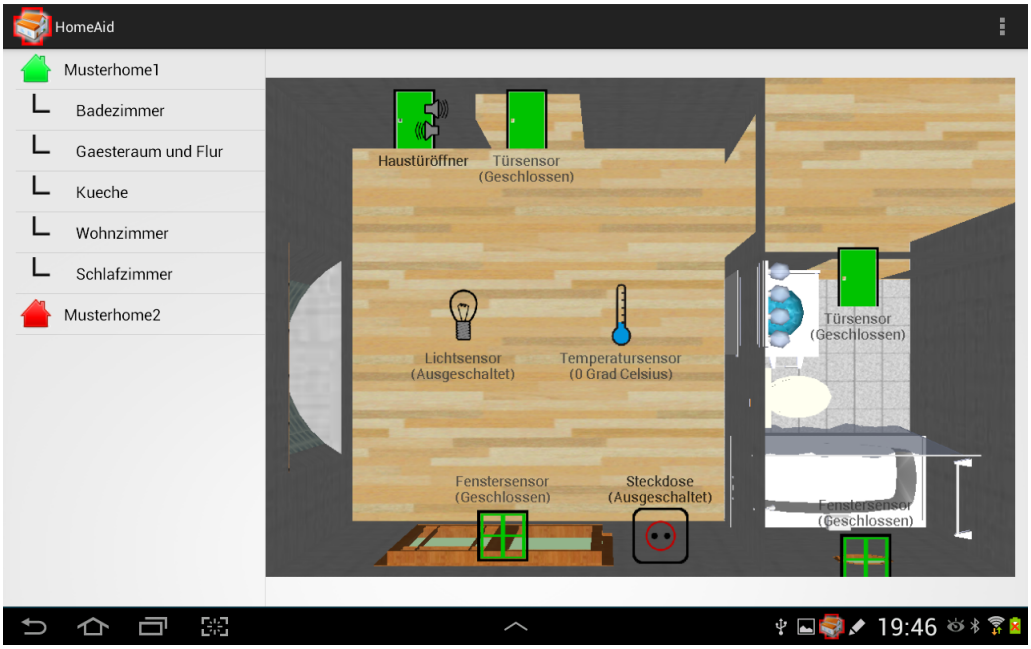


Abbildung 11: Screenshot Tablet

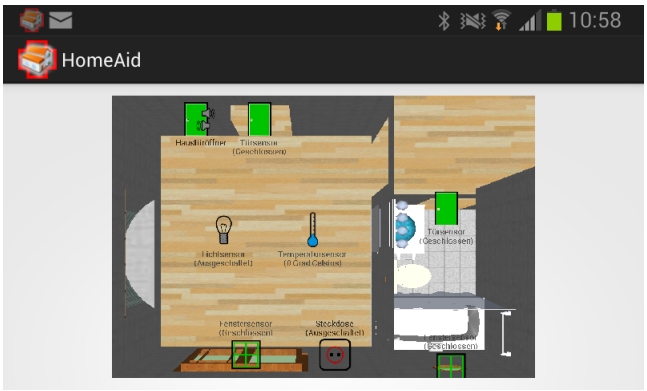


Abbildung 12: Screenshot Smartphone

4.2.2 ListViewFragment

Das *ListViewFragment* zeigt in einer Liste die Haushalte und ihre Räume an. Haushalte werden durch ein Haus-Symbol in der passenden Farbe zum Status (Verbunden=Grün, Nicht verbunden=Rot) angezeigt. Die Räume der jeweiligen Haushalte werden hierarchisch eingerückt angeordnet. In Abbildung 13 wird das Fragment auf einem Smartphone gezeigt. Ein langer Klick auf den Haushalt öffnet das Kontextmenü, von wo aus man manuell die „Notfall-Seite“ starten kann (siehe *EmergencyActivity*). Ein normaler Klick öffnet die Einstellungsseite *HomeSettingsFragement*. Ein Klick auf einen der Räume öffnet das entsprechende *ImageViewFragment*. Die Menü-Liste aktualisiert sich regelmäßig um eine aktuelle Ansicht der verbundenen Haushalte zu gewährleisten.

Abbildung 13: Screenshot ListViewFragment

4.2.3 ImageViewFragment

Das *ImageViewFragment* (Siehe Abbildung 14) stellt die Steuerzentrale für den Pflegebedürftigen dar. Hier werden die Raum-Bilder inklusive der jeweiligen Sensoren angezeigt. Weiter werden im Hintergrund eine Aktualisierung der Anzeige und das Senden von etwaigen Steuersignalen durchgeführt. Dazu ist es praktisch sogenannte *AsyncTasks* zu verwenden, da diese bereits eine interne Callback-Logik implementieren über deren *onPostExecute()*-Methode jeweils das Aktualisieren der Bildschirmseite durchgeführt werden kann. Das ist erforderlich, weil bei Android Zugriffe auf die Oberfläche nur über den Main-Thread der App erfolgen dürfen.

Die periodische Aktualisierung des Bildes und dessen Inhalt wird über ein *Timer*-Objekt gestartet, welches wiederum eine Instanz des besagten *AsyncTasks* startet. Die Anpassung der Periodendauer kann auf der App-Einstellungsseite vorgenommen werden. Beim Aufruf des Fragments wird die angeforderte Raum-Id über die Nummer des gedrückten Menüpunkts ermittelt. Über die Raum-Id wird das zugehörige *Image*-Objekt geladen und anschließend werden über dessen *refresh()*-Methode das Bild selbst, sowie die aktuellen Sensor-Werte und -Parameter vom Control-Server asynchron angefordert. Das Bild

wird im App-Verzeichnis mittels eindeutiger Kennung gespeichert und anhand des Hash-Wertes wird beim Neu-Laden verglichen, ob eine neue Übertragung notwendig ist. Die Methode *fillCurrentBitmapWithSensors()* lädt danach (wieder im Main-Thread) die dem Sensor-Status entsprechenden Icons, skaliert sie je nach Bildgröße und setzt sie auf die vom Control-Server festgelegte Position innerhalb des Bildes. Zusätzlich wird ein Text am Icon eingeblendet, der Bezeichnung und Status des Sensors beinhaltet. Sollte ein Icon keinen vom Control-Server bekannten Status besitzen wird ein rotes X über das Icon gelegt. Das Einblenden der Sensorgrafiken und der Statuseigenschaften wird durch eine Canvas-Fläche realisiert, die auf das Ursprungsbild gelegt wird und darauf diverse Zeichenmethoden zur Verfügung stellt. Anschließend wird das fertig gezeichnete Bild auf die aktuelle Bildschirmgröße skaliert und der ImageView-Container damit befüllt.

Um Akteure anzusteuern reicht eine Berührung von mindestens einer Sekunde auf das entsprechende Sensor-Symbol aus. Die Berührung selbst wird durch Implementierung der *onTouch()*-Methode im *OnTouchListener*-Interface der ImageView registriert. Um diese Berührung als Sensor-Berührung zu erkennen wurde eine Logik implementiert, die die Berührungs-Koordinate auf dem Bildschirm in eine reine Bildkoordinate umrechnet und mittels festgelegtem Radius vergleicht welcher Sensor sich darin befindet. Im Anschluss wird bei positivem Ergebnis auf gefundenen Sensor mit Akteur-Funktion wieder ein *AsyncTask* gestartet, der den Befehl übermittelt und eine aktualisierte Auflistung der Sensoren entgegennimmt. Bislang gibt es in der App noch keine eigene Klassifizierung für Akteure, die Akteur-Funktion stellt also lediglich eine Erweiterung der Sensoren dar. Die Gruppe dieser Quasi-Akteure ist augenblicklich auf die beiden „Sensoren“: Türöffner und Steckdosenschalter begrenzt.

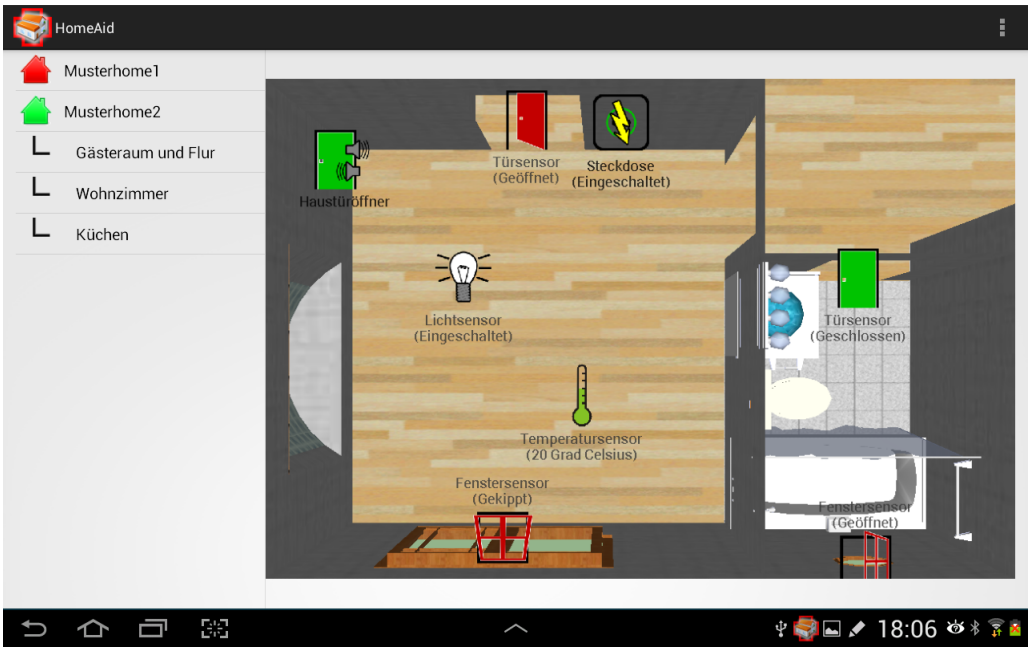


Abbildung 14: Screenshot ImageViewFragment in Aktion

4.2.4 HomeSettingsFragment

Das *HomeSettingsFragment* wird durch Klick auf die Haushalte aufgerufen. Hier werden die serverbezogenen Konfigurationsdaten eingetragen. Der betreffende Haushalt kann hier auch gelöscht werden. Jede durch *Speichern* oder *Löschen* bestätigte Änderung wird direkt angewendet und initiiert das Neu-Laden der *HomeAidActivity*. Die Einstellungen werden in Objekten der serialisierten Klasse *HomeConfig* gespeichert und durch *Java-Object-Streams* in lokaler Datei gespeichert bzw. aus ihr geladen. In Abbildung 15 wird die Oberfläche des *HomeSettingsFragment* gezeigt.

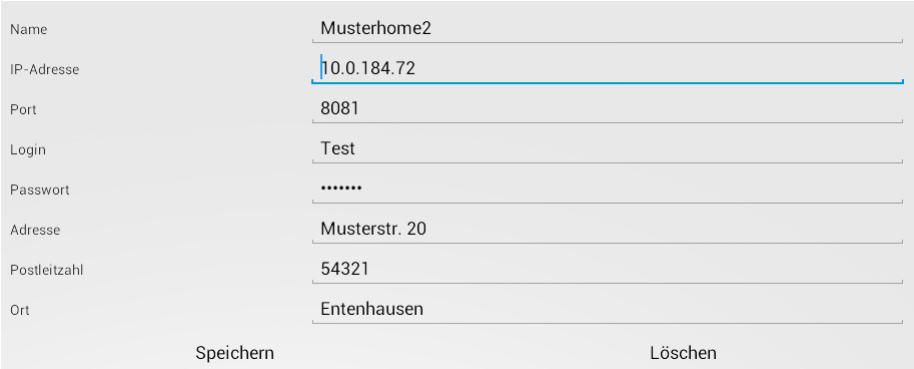


Abbildung 15: Screenshot HomeSettingsFragment

4.2.5 PreferencesActivity

Die Anpassungen der appweiten Einstellungen werden hier vorgenommen. Aus Kompatibilitätsgründen mussten die sog.*SharedPreferences* als Activity implementiert werden. Dabei handelt es sich um eine von *PreferenceActivity* abgeleitete Activity, die Anhand von vordefinierten Einstellungselementen innerhalb der Xml-View (Edit-Textfeld, Dropdown-Liste, Checkbox, ...) implizit mit Logik befüllt wird, ohne dass eine manuelle Weiterverarbeitung der Einstellungsdaten notwendig ist. In Abbildung 16 werden die möglichen Einstellungen gezeigt.

Abbildung 16: Screenshot PreferencesActivity

4.2.6 EmergencyActivity

Die Bildschirmseite *EmergencyActivity* (Abbildung 17) kann von zwei unterschiedlichen Ausgangspunkten aufgerufen werden. Zum Einen durch die Alarm-Notification und zum Anderen durch das Haushaltskontextmenü im *ListViewFragment*. Bei letzterem kann man zwischen „Zum Ziel navigieren“und „Haustür öffnen“wählen, während man bei der Alarm-Notification noch zusätzlichen die beiden Optionen erhält den akustischen Alarm zu unterbrechen und die Notification zu entfernen.

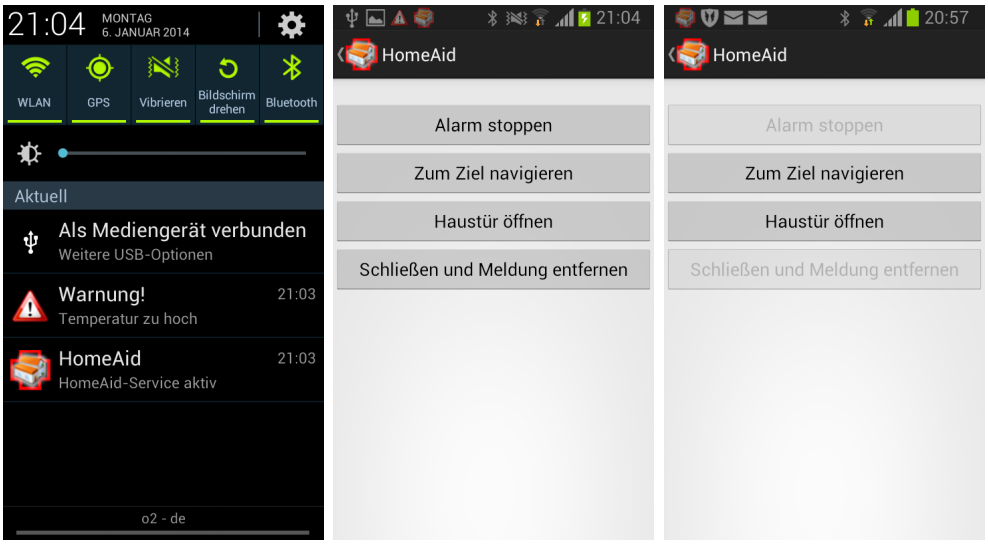


Abbildung 17: Screenshots EmergencyActivity bei automatischem und manuellem Start

4.3 Backend

Im Backend werden Hintergrund-Arbeiten durchgeführt: Es werden Verbindungen und Daten verwaltet und mittels Entity-Klassen-Struktur verknüpft. Die schematische Darstellung wird in Abbildung 18 aufgezeigt.

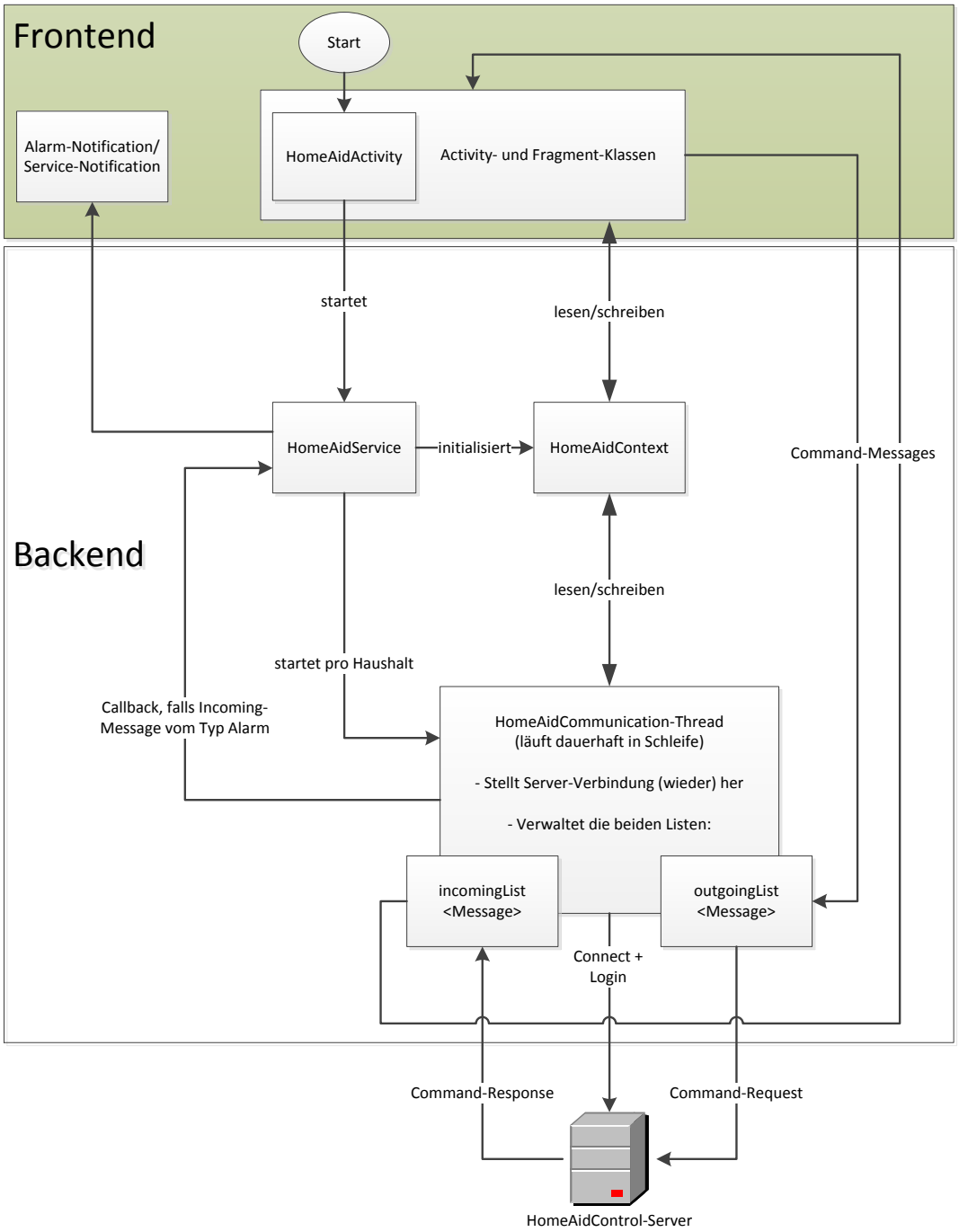


Abbildung 18: Struktur im Backend

4.3.1 HomeAidContext

Die Klasse *HomeAidContext* stellt einen Container mit Klassenvariablen und Methoden dar, dessen Inhalte innerhalb der gesamten App geteilt werden. Der wichtigste Inhalt bildet hier die Liste der Haushalt-Objekte, die im Gegensatz zu den sonstigen persistenten Einstellungen (Preferences) in einer seperaten Datei (*HomesList.dat*) gespeichert wird (Siehe *HomeSettingsFragment*). Die Methoden *saveHomes()*, *loadHomes()* und *clearHomes()* sind hierbei für Laden, Speichern und Aktualisieren der Liste verantwortlich.

4.3.2 HomeAidService

Der *HomeAidService* bildet das Zentrum für Hintergrund-Aktivität der App. Hier werden beim Start des Services die Werte des *HomeAidContext* initialisiert und anschließend zu jedem Haushalt(*Home*) ein *HomeAidCommunication*-Thread gestartet. Der Service erstellt eine System-Meldung (*Notification*), die dauerhaft über den Service-Status informiert. Der Service startet außerdem eine (oder mehrere) Alarm-Notifications, wenn vom Control-Server entsprechende Alarm-Ereignisse eingehen. Außerdem wird ein akustischer Alarm ausgelöst und bei Klick auf die jeweilige Notification eine Bildschirmseite (*Emegergency-Activity*) gestartet, die diverse Optionen anbietet um z.B. schnell per Google Maps zum Haushalt zu navigieren oder vor Ort die Tür öffnen zu können. Die minimal nötige Kommunikation von den Threads und der Emergency-Activity zum Service wird über Callback-Methoden gelöst.

4.3.3 HomeAidCommunication

Die Klasse *HomeAidCommunication* erbt von der Klasse *Thread* und implementiert somit die Methode *run()*, die in einem neuen Thread ausgeführt wird. Diese Methode läuft dauerhaft in einer Schleife und prüft für den jeweiligen Haushalt, ob die Verbindung besteht und stellt sie gegebenenfalls (wieder) her. So wird gewährleistet, dass die Verbindung zum Control-Server dauerhaft besteht, soweit die Rahmenbedingungen (Internetverbindung, Empfang) dies zulassen. Die konstante Verbindung ist sowohl für eingehende Alarm-Nachrichten als auch für die automatische Aktualisierung von Sensorwerten wichtig. Außerdem werden von hier die Nachrichten-Warteschlangen in Form der beiden Listen *incomingList* und *outgoingList* verwaltet. Diese wurden als Abstraktion von Array-Listen implementiert, zuzüglich der benötigten Methoden unter Berücksichtigung von synchronisiertem (Thread-sicherem) Zugriff. Diese Listen beinhalten *Message*-Objekte, die sich aus Erstellungszeitpunkt, Typ, Payload und einem Flag, ob auf eine Antwort-Nachricht gewartet werden soll, zusammensetzen. Die zur Klasse *HomeAidCommunication* zugehörige Instanzmethode *sendCommand(Command cmd)* nimmt ein der Klasse *Command* abgeleitetes Objekt entgegen und führt die jeweils überladenen Ein- und Ausgabeprozeduren aus, indem sie die Befehle in *Message*-Objekte verpackt, in die *outgoingList* einreiht, auf neue Nachrichten innerhalb der *incomingList* wartet, die nächste zugehörige Nachricht entpackt und als erwartetes Objekt zurück gibt.

4.3.4 HomeAidSocket

Die Klasse *HomeAidSocket* stellt eine angepasste Abstrahierung der Klasse *SSLSocket* dar und implementiert Methoden zur Verbindungsverwaltung. Die Methode *connect()* initiiert die SSL-Verbindung zur Middleware, startet den Handshake und akzeptiert dabei durch die *NaiveThrustManager*-Klasse [5] jedes Serverzertifikat, da der Fokus hier lediglich auf der verschlüsselten Übertragung liegt. Während die *write(Message msg)*-Methode einfach

die gleichnamige Methode der *SSLSocket*-Instanz aufruft und den Inhalt des *Message*-Objektes weitergibt, ist die *read()*-Methode etwas komplexer umgesetzt. Aufgrund der Ungewissheit, wie viele Daten ankommen wird hier ein *ByteArrayOutputStream*-Objekt erzeugt, in einer Schleife befüllt solange Daten ankommen und letztlich in ein *Message-Objekt* gepackt. Weitere Methoden innerhalb dieser Klasse realisieren den Login, Disconnect und einen Ping-Befehl.

4.3.5 Command-Klassen

Für jedes Kommando, das der Control-Server kennt gibt es hier eine eigene *Command*-Klasse. Je nach Kommando werden beim Instanzieren durch den Konstruktor noch Parameter erwartet. Diese Klassen implementieren das Interface *Command* und somit die Methoden *getCommandMessage()* und *getReceivedObject(Message msg)*. Erstere liefert ein *Message*-Objekt zurück, dass den fertig formulierten Bytecode für das entsprechende Kommando zum Senden enthält. Letztere Methode interpretiert das empfangene *Message*-Objekt (bzw. dessen Payload) und erstellt daraus das jeweilige Objekt zur Rückgabe. Für jedes Kommando, das der Control-Server kennt gibt es hier eine eigene *Command*-Klasse. Je nach Kommando werden beim Instanzieren durch den Konstruktor noch Parameter erwartet. Diese Klassen implementieren das Interface *Command* und somit die Methoden *getCommandMessage()* und *getReceivedObject(Message msg)*. Erstere liefert ein *Message*-Objekt zurück, dass den fertig formulierten Bytecode für das entsprechende Kommando zum Senden enthält. Letztere Methode interpretiert das empfangene *Message*-Objekt (bzw. dessen Payload) und erstellt daraus das jeweilige Objekt zur Rückgabe. Wie bereits beschrieben werden können *Command*-Objekte an die *sendCommand(Command cmd)*-Methode übergeben werden, wo das Kommando ganzheitlich bearbeitet wird. Die Kommandos in Abbildung 19 wurden bisher implementiert.

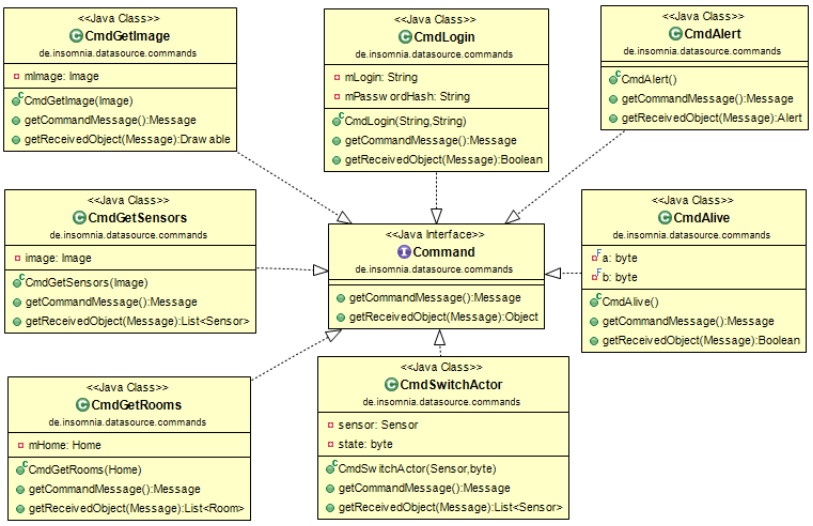


Abbildung 19: Kommandos

4.3.6 Entitys

Die Entitäten setzen sich aus den Klassen zusammen, die je eindeutige Objekte zur Datenmodellierung innerhalb der App bilden. Durch die Verknüpfungen der Entitys untereinander wird eine hierarchische Struktur ausgehend von *Home*, über *Room* und *Image* bis hin zu verschiedenen Versionen von *Sensor* umgesetzt. Das Diagramm in Abbildung 20 zeigt die zugrunde liegende Struktur.

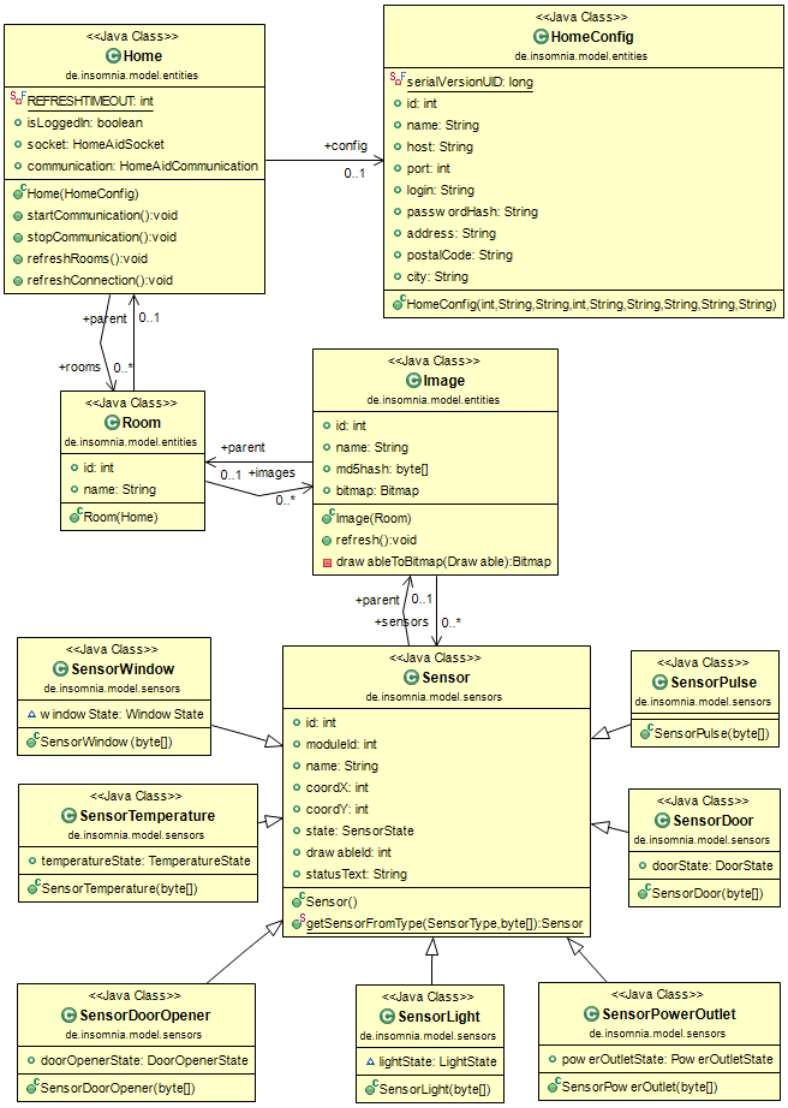


Abbildung 20: Entity Beziehungen

4.4 Metriken

Diverse Kennzahlen der App werden in Abbildung 21 gezeigt. Erstellt wurden diese durch das Eclipse-Plugin *Metrics*.

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods (avg/max per type)	10	0,213	0,409	1
Number of Attributes (avg/max per type)	98	2,085	2,887	11
Number of Children (avg/max per type)	2	0,043	0,289	2
Number of Classes (avg/max per packageFragment)	47	5,222	4,157	12
Method Lines of Code (avg/max per method)	1506	10,176	13,936	103
Number of Methods (avg/max per type)	141	3	3,73	19
Nested Block Depth (avg/max per method)		1,662	0,969	6
Depth of Inheritance Tree (avg/max per type)		1,638	1,312	7
Number of Packages	9			
Afferent Coupling (avg/max per packageFragment)		9,889	8,987	27
Number of Interfaces (avg/max per packageFragment)	4	0,444	0,685	2
McCabe Cyclomatic Complexity (avg/max per method)		2,304	1,972	15
Total Lines of Code	2920			
Instability (avg/max per packageFragment)		0,33	0,307	1
Number of Parameters (avg/max per method)		0,865	1,107	9
Lack of Cohesion of Methods (avg/max per type)		0,15	0,301	0,917
Efferent Coupling (avg/max per packageFragment)		3,444	3,37	10
Number of Static Methods (avg/max per type)	7	0,149	0,545	3
Normalized Distance (avg/max per packageFragment)		0,572	0,335	1
Abstractness (avg/max per packageFragment)		0,171	0,311	1
Specialization Index (avg/max per type)		0,102	0,24	1,2
Weighted methods per Class (avg/max per type)	341	7,255	9,888	38
Number of Static Attributes (avg/max per type)	174	3,702	12,071	71

Abbildung 21: Metriken der Android-App (Stand: 10.01.2014)

4.5 In Planung

Folgende Features sind für die Zukunft geplant:

- Rechteverwaltung für unterschiedliche Arten von Nutzern
- Zeitsteuerung für das Senden von Befehlen an Akteure
- Mehrere Bilder pro Raum

5 Weboberfläche

5.1 Funktionalität

Die Web-Applikation der HomeAid Software soll als Ergänzung zur Android-App benutzt werden und es ermöglichen die Funktionen bequem von jedem PC mit Internetzugang aus benutzen zu können. Das Web-Interface soll sehr einfach zu bedienen sein, um es auch älteren Menschen zugänglich zu machen. In der Web-Applikation können Bilder zu den einzelnen Räumen abgefragt werden, die wiederum Informationen über die Sensoren enthalten. Wie bei der Android-App auch kann man so im Haushalt sehen ob Steckdosen oder Glühbirnen angeschaltet, beziehungsweise Fenster oder Türen offen sind. Diese Abfrage erfolgt durch die Verwendung eines selbst definierten Protokolls, über welches Befehle an den Server übertragen werden und von ihm empfangen und ausgewertet werden.

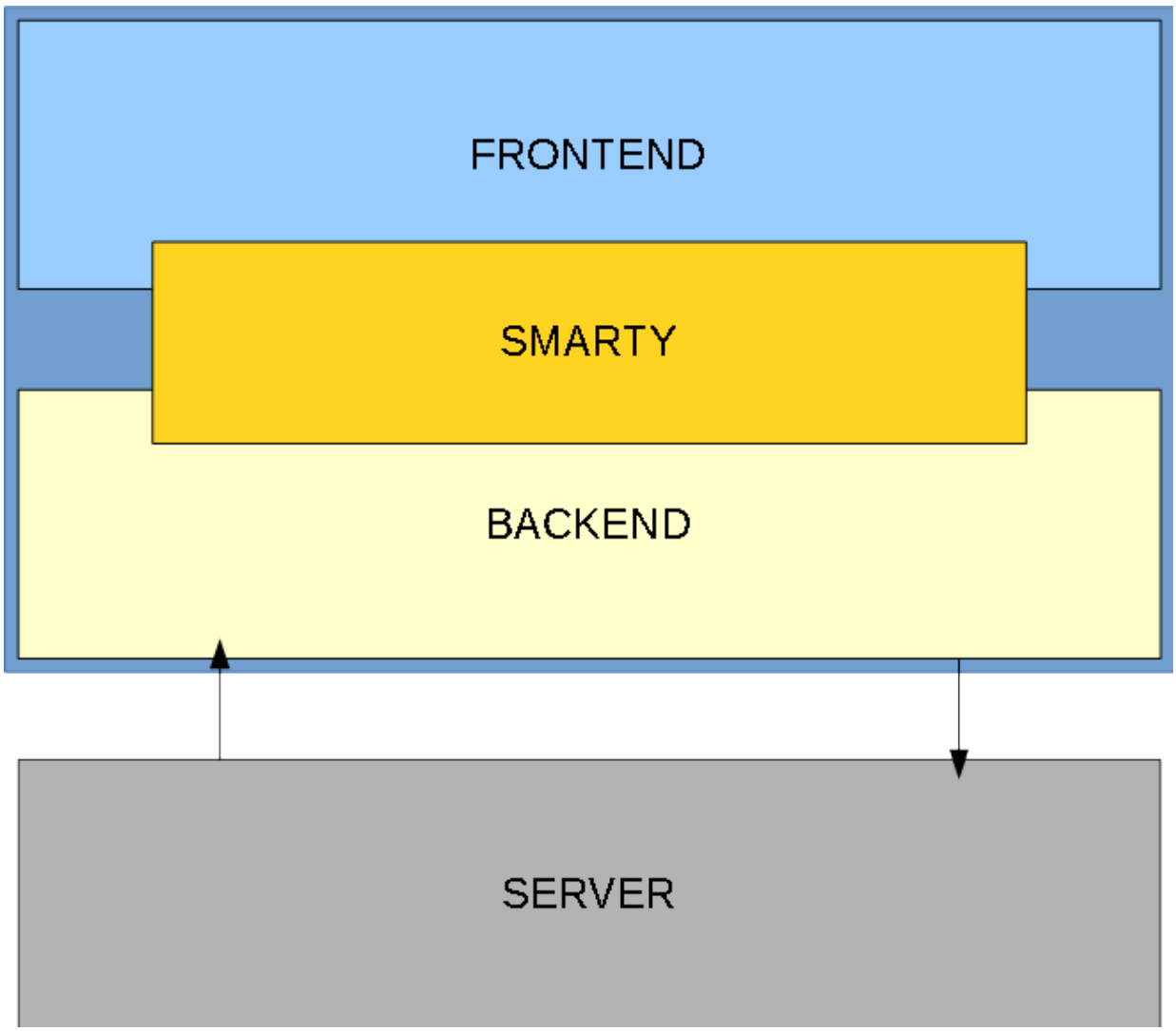


Abbildung 22: Übersicht Weboberflaeche

5.2 Frontend

Das Frontend, also die Benutzeroberfläche, ist so einfach wie möglich gehalten. Sie beschränkt sich auf eine Navigationsleiste am oberen Bildschirmrand, die ausschließlich zum registrieren, einloggen oder - wenn bereits eingeloggt - anzeigen der Räume dient und einer dynamisch erstellten Liste aller für den eingeloggten Benutzer registrierten Räume, welche dann ausgewählt und angezeigt werden können.
Das gesamte Frontend wurd unter Zuhilfenahme von *Twitter Bootstrap 3* gestaltet.
Außerdem wurde zur Trennung der Anwendungslogik von der Präsentation die Template Engine *Smarty* benutzt.
Das folgende Bild ist ein Ausschnitt eines Screenshots des HomeAid Indexes.

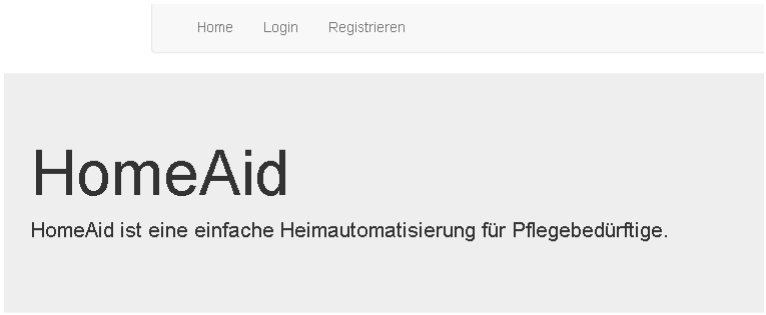


Abbildung 23: Ausschnitt Oberfläche

5.2.1 Twitter Bootstrap 3

Bootstrap ist ein Framework das Style Templates zur einfachen gestaltung von Weboberflächen enthält. Bootstrap besteht hauptsächlich aus Cascading Style Sheets (CSS). Es stellt außerdem ein Grid-System zur Verfügung, welches die angezeigte Seite in Grid-Segmente aufteilt, die individuell angesprochen, formatiert und mit Inhalten gefüllt werden können. Bootstrap ist mit Stylesheets ausgestattet, die für alle wesentlichen HTML-Komponenten Stildefinitionen enthalten. Zusätzlich zu den bestehenden, regulären HTML-Elementen enthält es viele Oberflächenelemente, die häufig verwendet werden (wie Buttons, Navigationsleisten, Labels usw.). Es folgt ein Beispiel, welches die Verwendung einer Navigationsleiste zeigt.

```
<!DOCTYPE html>
<html>
  <head>
    <title>HomeAid{if isset($title)} - {$title}{/if}</title>
    <link rel="stylesheet" type="text/css" href="templates/css/bootstrap.css">
    <link rel="stylesheet" href="templates/css/signin.css">
  </head>
  <body>
    <div class="container"> <!-- PARENT CONTAINER -->

      <!-- NAVBAR -->
      <div class="navbar navbar-default" role="navigation">
        <div class="container">
          <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav" id="nav">
              <li><a href="index.php">Home</a></li>
              {if isset($isLoggedIn)}
                <li><a href="index.php?action=manageRooms">Rooms</a></li>
                <li><a href="index.php?action=logout">Logout</a></li>
              {else}
                <li><a href="index.php?action=login">Login</a></li>
                <li><a href="index.php?action=register">Registrieren</a></li>
              {/if}
            </ul>
          </div>
        </div>
      </div>
    </div>
```

Abbildung 24: Bootstrap Beispiel

5.2.2 Smarty

Smarty ist eine Template Engine in Form einer PHP-Bibliothek und ermöglicht es sehr einfach den Code einer Webanwendung von deren Anzeige zu trennen. Es liest Template Dateien aus und generiert daraus neue PHP-Skripte, die nicht bei jedem Aufruf der Seite neu geparst werden müssen. Hier ein Beispiel zur Index Seite der HomeAid-Software. Die Zeilen 1 und 6 ({*include file='FILENAME.SUFFIX'*}) fügen an den Stellen, an den sie stehen, andere Templates ein, die wiederum eigenen HTML Code enthalten. Der Header fängt mit dem öffnenden HTML-Tag an und der Footer hört mit dem schließenden HTML-Tag auf. So ist es sehr leicht machbar mehrere mögliche Webseiten zu ändern, ohne viel am Code umschreiben zu müssen.

```
{include file='header.tpl'}
<div class="jumbotron">
<h1>HomeAid</h1>
<p class="lead">HomeAid ist eine einfache Heimautomatisierung fuer Pflegebeduerftige.</p>
</div>
{include file='footer.tpl'}
```

Abbildung 25: Smarty Beispiel

5.3 Backend

Das Backend wurde ausschließlich in PHP programmiert und durch Smarty mit dem Frontend verbunden. Insgesamt gibt es zwei Hauptdateien:
Zum einen die *index.php*-Datei, in welcher über die GET-Methode Informationen als Parameter in der URL gelesen und verarbeitet werden. Über eine switch-Anweisung führen die verschiedenen Parameter dann zu unterschiedlichen Operationen - meistens Funktionsaufrufe.
Zum anderen die *manageRooms.php*-Datei, die sämtliche Funktionen enthält und Daten vom Server empfängt, diese verarbeiten kann und Befehle an den Server schicken kann.
Die Befehle werden byteweise übertragen. Die wesentlichen Befehle sind: *connect()* um eine Verbindung zum Server aufzubauen, *read_data()* um die vom Server gesendeten Daten zu lesen, *get_rooms()* um eine Liste der im Home registrierten Räumen zu erhalten, *get_image_info()* sowie *get_image()* um zuerst Informationen zu einem Bild abzurufen und dann das entsprechende Bild vom Server anzufordern und schließlich *draw_image()* um das angeforderte Bild mit den Informationen zu füllen und anzuzeigen.

5.3.1 read_data()

Die read_data()-Funktion wird in fast jeder anderen Funktion aufgerufen. Sie ist dafür da, die vom Server empfangenen Bytes zu lesen, in einem Array zu speichern und an die aufrufende Funktion zurückzugeben. Da in PHP maximal 8192 Bytes auf einmal gelesen werden können, muss in einer Schleife ein Buffer realisiert werden, in den bei jedem Schleifendurchlauf - falls die Nachricht dies erfordert - 8192 Bytes Informationen geschrieben werden. Der folgende Codeausschnitt zeigt die entsprechende Funktion.

```
private function read_data() {
  $r = array($this->fp);
  $null = NULL;
  $read = "";
  $tryNextLoop = TRUE;

  while($tryNextLoop && ($num_changed_streams =
                stream_select($r, $null, $null, 0, 1000000)) != false) {
    if($num_changed_streams > 0) {
      $buf = fread($this->fp, 8192);
      $tryNextLoop = (strlen($buf) == 8192);
      $read .= $buf;
    }
    else {
      if($this->fp)
        fclose($this->fp);
      return FALSE;
    }
  }

  // Error
  if(strlen($read)<=0) {
    $this->smarty->assign('error', "");
    $this->smarty->assign('errorMsg', "Reading failure.
                        Could not read received data. read_data");
    if($this->fp)
      fclose($this->fp);
    return FALSE;
  }
  return $read;
}
```

Abbildung 26: Codebeispiel read_data()

5.3.2 get_rooms()

In dieser Funktion wird beim Server abgefragt, wie viele Räume für den eingeloggten Benutzer registriert sind, wie sie heißen und welche Bild-IDs zu den Räumen gehören. Diese ganzen Informationen werden dann in einem Array abgelegt, welches dann zurückgegeben wird. Die angeforderten Räume tauchen dann auf der Website als Menü auf. Bei einem Klick auf den Namen eines Raums wird dann das dazugehörige Bild mit samt seinen Informationen angefordert und angezeigt. Aufgerufen wird die Funktion automatisch mit dem Klick auf den Menüpunkt „Rooms“ in der Weboberfläche. Im folgenden Bild sieht man einen Ausschnitt der Webseite in dem die Räume aufgelistet werden. (Im Beispiel ist nur ”Prototyp Platte”registriert)

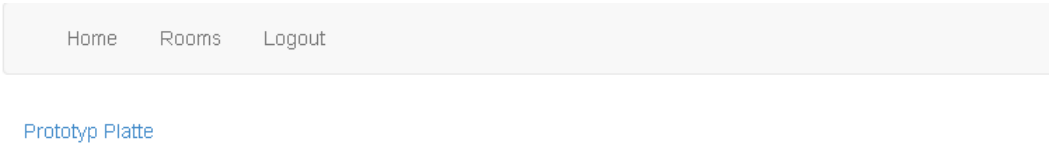


Abbildung 27: Ausschnitt Oberfläche

5.3.3 get_image_info() und get_image()

Die beiden Funktionen get_image_info() und get_image() laufen Hand in Hand. Zunächst wird in der Weboberfläche durch einen Klick auf einen Raumnamen im Menü die Funktion

get_image_info() aufgerufen. Die Bild-ID, die als Parameter an die Funktion übergeben wird, wurde vorher über get_rooms() abgefragt und gespeichert und nun als Parameter in der URL übergeben. Innerhalb der get_image_info()-Funktion wird nun ein Image-Objekt angelegt. Anschließend wird die get_image()-Funktion aufgerufen um die Bildinformationen als Byte-Array vom Server anzufordern und in dem Image-Objekt in der *drawable*-Variablen gespeichert. Zurück in der get_image_info()-Funktion wird nun überprüft wie viele Sensoren es in diesem Raum - also auch auf dem Bild - gibt und es werden Informationen zu diesen (Typ, ID, Koordinaten und Status) angefordert. Diese Sensorinformationen werden dann in dem erstellten Image-Objekt gespeichert und das Objekt selbst wird schließlich als Rückgabewert zurückgegeben. Dieses Objekt enthält nun genug Informationen um angezeigt werden zu können, allerdings müssen zuerst noch die Sensoren an ihre entsprechenden Stellen platziert und je nach Status durch ein anderes Icon repräsentiert werden.

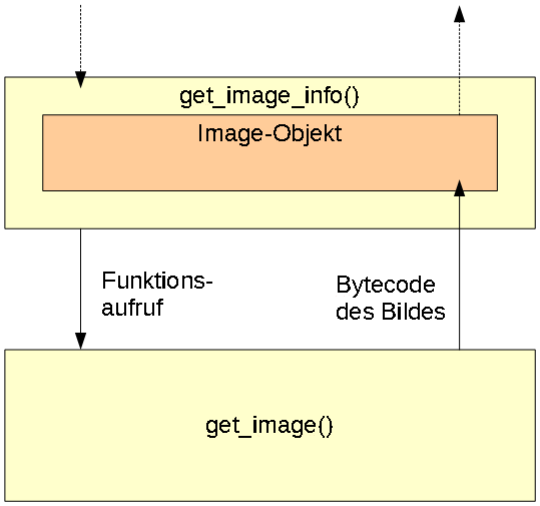


Abbildung 28: Funktionsweise der beiden Image-Funktionen

5.3.4 draw_image()

In dieser Funktion wird das über get_image_info() angeforderte und mit Sensorinformationen gefüllte Bild mit seinen Sensoren gespickt. Zunächst werden durch eine switch-Anweisung die Sensoren des Image-Objektes in einer Schleife durchiteriert und mittels der imagecopyresized()-Funktion auf das ursprüngliche Bild eingefügt. Mit jedem Schleifendurchlauf wird also überprüft, ob es noch Sensoren gibt und - falls ja - welchen Typ und welchen Status diese haben, um dann am Ende der Schleife das entsprechende Icon auf das Bild zu kopieren. Nachdem dann alle Sensoren auf das Bild kopiert wurden wird dieses in den Output-Buffer geschoben, um ihn als String auslesen zu können und am Ende der Funktion - base64 encoded - als globale Smarty-Variablen in der manageRooms.tpl sichtbar zu machen und anzeigen zu lassen.

6 Zusammenfassung

Bei der Umsetzung des Projekts „Heimautomatisierung für Pflegebedürftige“ wurde besonderer Augenmerk auf größtmögliche Flexibilität im Bezug auf die verwendbare Hardware gelegt. Daher besteht das Projekt aus den vier Komponenten Hardware-Prototyp, Middleware, Android-App und Weboberfläche. Die Middleware bildet die zentrale Schnittstelle zwischen Hardware und höheren Anwendungen. Je nach Hardware kann hier dynamisches Laden des entsprechenden Treiber-Moduls stattfinden. Der Prototyp steht stellvertretend für eine beliebige Heimautomatisierungs-Hardware und stellt die wichtigsten Funktionen derer bereit. Die zugehörige Android-App stellt Kontroll- und Schaltfunktionen zur übersichtlichen Verwaltung via Tablet und Smartphone zur Verfügung. Über die Weboberfläche lassen sich zusätzlich plattformunabhängig die Sensorwerte einsehen.

Das Projekt konnte innerhalb des gesetzten Zeitrahmens fertig gestellt und optimiert werden. Allerdings wurde es durch den weitreichenden Umfang des Projekts nötig, dass die Teammitglieder bereitwillig mehr Zeit investierten, als ursprünglich geschätzt.

Abschließend betrachtet, hat das Projekt die Team-Mitglieder um viele Erfahrungen bereichert. Die Mitglieder hatten durch die Bearbeitung des jeweiligen Teilgebiets eine eigene Herausforderung zu bewältigen, während sie zusätzlich durch die enge lokale Zusammenarbeit vom Erfahrungsschatz der anderen Teammitglieder profitieren konnten. Auch waren viele allgemeine Absprachen notwendig um das optimale Zusammenspiel der Teilkomponenten zu erreichen.

Literatur & Quellen

[1] Homepage RapidXml
http://rapidxml.sourceforge.net/ (06.01.2014)

[2] Abbildung Pegelwandler
http://www.exp-tech.de/images/product_images/info_images/8channellevelconverter_1.jpg/
http://www.exp-tech.de/Shields/8-channel-Bi-directional-Logic-Level-Converter—TXB0108.html

[3] Ansteuerung der Fernbedienung
http://www.knackes.com/blog/index.php/2011/08/ardomo-domotique-hx2262-cny74-4-ethernet/

[4] Becker, Arno & Pant, Marcus: *Android 4.4: Programmieren für Smartphones und Tablets - Grundlagen und fortgeschrittene Techniken*

[5] Accepting Self-Signed SSL Certificates in Java
http://www.howardism.org/Technical/Java/SelfSignedCerts.html