Fachhochschule Düsseldorf

SIEMENS

Bachelorthesis

Einbindung und Optimierung einer Heizungs-regelung mittels eines Software-Frameworks in das Funknetzwerk des Ambient Assisted Living – Forschungsprojekts

Integration and optimization of a heating Control via a software framework into the wireless network of the Ambient Assisted Living research project

Erstellt durch
Name: **Daniel Andreas**
Matrikelnummer: **548398**

1.Prüfer: **Prof. Dr.-Ing. Ulrich G. Schaarschmidt**
2.Prüfer: **Prof. Dr. rer. nat. Wolfgang Lux**

Düsseldorf, den 07.08.2013

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig verfasst habe.

Ich versichere, dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahren gewesen ist.

_____

Daniel Andreas

FH D SIEMENS

Danksagung

Zuerst möchte ich mich bei meinen beiden betreuenden Professoren, Herrn Prof. Dr.-Ing. Ulrich G. Schaarschmidt und Herrn Prof. Dr. rer. nat. Wolfgang Lux bedanken, die mir immer mit Rat und Tat zur Seite standen.

Weiterhin möchte ich mich bei den MitarbeiterInnen des Informatiklabors der Fachhochschule Düsseldorf bedanken. Tanja Kleiner gab mir hilfreiche Tipps, wenn *OSGi* sich nicht so verhielt wie Bücher oder meine Vorstellung es suggerierten, Wolfram Gerlach half mir bei den Hardwareproblemen, vor allem der Fehlersuche bei der Funkkommunikation, Mirco Kern bei *Contiki* und *OSGi* im Allgemeinen, Marc Rompa wies mich auf Probleme beim Schreibzugriff auf das *EEPROM* hin und Oliver von Fragstein hatte immer ein offenes Ohr für alles andere.

Last but not least möchte ich mich bei meinen Eltern bedanken, die mich immer unterstützten und akzeptierten, dass ich zwölf Wochen lang nur an die Bachelorarbeit denken konnte.

Inhaltsverzeichnis

1 Einführung

Die Bachelorthesis entstand im Rahmen des *WieDAS*-Forschungsprojekts im Informatiklabor des Fachbereichs Elektrotechnik der Fachhochschule Düsseldorf. *WieDAS* ist ein Akronym für **Wie**sbaden-**D**üsseldorfer **A**mbient Assisted Living **S**ervice Plattform, ein gemeinsames Forschungsprojekt der Hochschule RheinMain in Wiesbaden und der Fachhochschule Düsseldorf, welches sich mit der Erforschung und Umsetzung von Unterstützungssystemen im Rahmen des Ambient Assisted Livings befasst.

„Unter „Ambient Assisted Living" (*AAL*) werden Konzepte, Produkte und Dienstleistungen verstanden, die neue Technologien und soziales Umfeld miteinander verbinden und verbessern mit dem Ziel, die Lebensqualität für Menschen in allen Lebensabschnitten, vor allem im Alter, zu erhöhen.“
(Bundesministerium für Bildung und Forschung)

Im Rahmen des *WieDAS* Forschungsprojektes wurden so unter anderem ein Notfallknopf, der erkennt, ob derjenige, der ihn trägt, gestürzt ist, ein Wassermelder, der Alarm schlägt, wenn er Feuchtigkeit detektiert und eine Türöffnung mit Webcam realisiert.

Während meines Praxisprojektes entwickelte ich im Umfeld des *AAL* eine Heizungsregelung, da die auf dem Markt erhältlichen Geräte nicht über das im Labor eingesetzte *6LoWPAN*-Funkprotokoll kommunizieren und sie keine frei verfügbaren Programmierschnittstellen besitzen.

Die Bachelorthesis erweitert die Heizungsregelung nun um eine, mit dem eingebauten Touch Panel bedienbare, Benutzeroberfläche und bindet sie in das bestehende Softwareframework ein, mit dem die anderen, bereits vorhandenen, Geräte über einen PC bedient und beobachtet werden können. Das zu diesem Zweck eingesetzte Framework ist ein *OSGi*-Framework, ein Modulsystem für Java, bei dem Softwarekomponenten dynamisch zur Laufzeit hinzugefügt werden können.

Zunächst folgt eine Bestandsaufnahme der im Praxisprojekt entwickelten Hard- und Software.

2 Bestandsaufnahme

Die im Rahmen der Bachelorthesis zu optimierende Heizungsregelung entstand im Informatiklabor der Fachhochschule Düsseldorf während meines Praxisprojekts (Andreas, 2013). Zunächst wird auf die erstellte Hardware, welche in Abbildung 1 zu sehen ist, eingegangen und anschließend auf die Software.

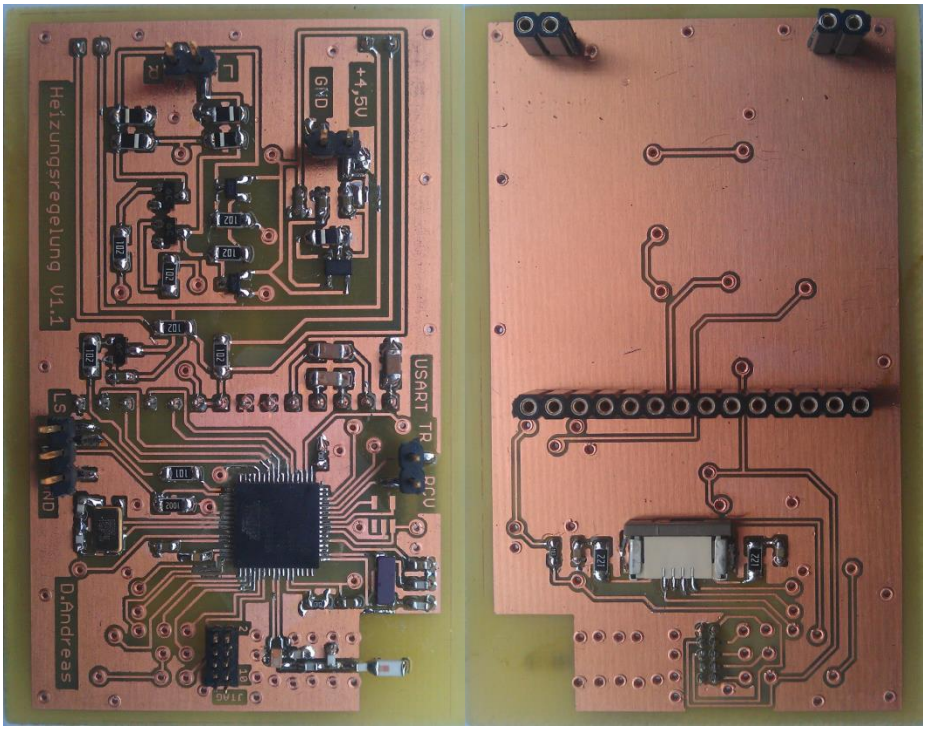


Abbildung 1: Praxisprojekt-Schaltung

Die Hardware der Heizungsregelung besteht aus:

- einem Mikrocontroller vom Typ *ATmega128RFA1* der Firma Atmel,
- einem digitalen Temperatursensor vom Typ *TC74* von Microchip,
- einem LC-*Display DOGS102W-6* von Electronic Assembly,
- einer weißen LED-Hintergrundbeleuchtung vom Typ *LED39x41-W* von Electronic Assembly,
- einem resistiven Touch-Panel vom Typ *TOUCH102-1*, von Electronic Assembly,
- einem 3,3V Festspanungsregler vom Typ *LP2985* von Texas Instruments,
- einer 2,45 GHz Keramik Antenne von Johanson Technology,
- sowie einer H-Brücke für die Motoransteuerung.

Um auf die Heizung einwirken zu können, kommt ein aus dem Heizkörperthermostat *K92457* von eQ-3 ausgebauter Gleichstrommotor zum Einsatz. Dieser treibt ein Getriebe mit einer starken Untersetzung an, welches über ein Schraubgetriebe den Ventilstift verfährt. Motor und Getriebe sind in Abbildung 2 zu sehen. Die Programmierung des Mikrocontrollers erfolgte über das **J**oint **T**est **A**ction **G**roup (*JTAG*)-Interface, welches das Programmieren und das Debugging des Mikrocontrollers in der Schaltung ermöglicht.

Abbildung 2: Motor und Getriebe

Die realisierte Software nutzt das Open-Source Betriebssystem *Contiki*, welches speziell für Systeme mit geringer Speicherkapazität entworfen wurde. Die Software beinhaltet einen Regler, welcher, mithilfe der Treiberroutinen, die während des Praxisprojekts für den Temperatursensor geschrieben wurden, die Temperatur erfasst und mit dem Motor auf das Heizungsventil einwirkt, um die Temperatur zu regeln. Außerdem werden auf dem Display Debugginginformationen ausgegeben, wie die erfasste Temperatur, der erfasste Berührungspunkt des Touch Panels, die Ausgangsgröße des Reglers und der Sollwert für den Motor. Das Empfangen von Daten per *6LoWPAN* ist nicht möglich, die erfasste Temperatur wird allerdings via *6LoWPAN* als *UDP-Broadcast* verschickt.

Im folgenden Kapitel wird zuerst auf die noch bestehenden Probleme des Praxisprojekts eingegangen und anschließend auf Basis dieser und des Projektkontexts eine Zielsetzung formuliert.

3 Problemanalyse und Zielsetzung

Die Zielsetzung der Bachelorarbeit ergibt sich einerseits aus den Fehlern der bestehenden Hard- und Software und deren noch unvollständigen Funktionen und andererseits aus dem Kontext des WieDAS Forschungsprojektes.

Die vorhandene Schaltung hat, wie in der Dokumentation des Praxisprojektes (Andreas, 2013 S. 40) beschrieben, das Problem, dass die Abwärme des Mikrocontrollers den Temperatursensor über die Raumtemperatur erhitzen konnte. Das Layout wurde schon während des Praxisprojekts dahingehend angepasst, dass der Temperatursensor sich nicht mehr in der Nähe des Mikrocontrollers befindet. Außerdem soll die H-Brücke ausgetauscht werden, da sich diese als recht instabil herausstellte. Die abgeänderte Hardware soll anschließend neu aufgebaut werden.

Um die Einstellung der Solltemperatur des Thermostats zu ermöglichen, sowie Statusinformationen anzuzeigen, ist eine Graphische Benutzeroberfläche (**G**raphical **U**ser **I**nterface - *GUI*) zu erstellen und die Ansteuerung der Heizungsregelung per Funk zu ermöglichen. Nach einiger Zeit im Betrieb kann es außerdem passieren, dass einzelne *Protothreads* bei *Contiki* „einfrieren“. Die Ursache hierfür soll gesucht und behoben werden.

Nach der Fertigstellung dieser Komponenten erfolgt eine Einarbeitung in das *OSGi*-Framework, mit welchem die Software läuft, die die bereits vorhandenen Geräte des *WieDAS*-Forschungsprojekts ansteuert. Es soll ein eigenes Softwarepacket für die Heizungsregelung geschrieben werden, mit welchem beliebig viele Heizungsregelungen benutzerfreundlich verwaltet und bedient werden können.

Die praktische Umsetzung der Bachelorthesis wird im Folgenden beschrieben.

4 Umsetzung

Zunächst wird der Ablauf des Projektes kurz zusammengefasst, die ausführlichen Beschreibungen und Erklärungen sind in den folgenden Unterkapiteln enthalten. Wie im vorherigen Kapitel beschrieben, sollte zunächst die Hardware neu aufgebaut werden. Dazu wurde zuerst die H-Brücke durch ein bereits im Labor vorhandenes Motortreiber-IC ausgetauscht, da diese sich im Praxisprojekt als sehr unzuverlässig erwiesen hatte.

Der Schaltplan und das Layout wurden mit dem CAD-Programm *EAGLE* (**E**infach **An**zuwendender **G**rafischer **L**ayout-**E**ditor) der Firma CadSoft[1] erstellt und befinden sich auf der beigefügten CD. Im Labor wird ausschließlich *EAGLE* für diese Zwecke eingesetzt, außerdem es gibt eine kostenlose Version, mit der maximal eine halbe Europlatine (Ausmaße einer Europlatine: 160 mm x100 mm) designt werden kann. Die letztendlich entworfene Schaltung hat die gleichen Ausmaße (76mm x 45mm) wie die des Praxisprojekts, sodass diese Einschränkung für die Heizungsregelung unerheblich ist. Nach dem Entwurf des Schaltplans wird anhand dessen ein Layout entworfen, welches im Gegensatz zum abstrahierten Schaltplan, der lediglich die Symbole und Werte der Bauteile enthält, die realen Bauformen der Bauteile enthält. Nachdem diese alle platziert und mit Leiterbahnen verbunden sind, druckt man die (in diesem Fall) Oberseite und Unterseite des erstellten Layouts auf eine transparente Folie. Die eigentliche Platine besteht aus Kunststoff mit einer darauf haftenden Kupferschicht, die in diesem Fall 35 µm Dicke hat. Die Kupferschicht ist wiederum mit einem Fotolack überzogen, der sich bei einer Bestrahlung mit UV-Licht und anschließendem Bad in einem Entwickler aus Natriumhydroxid auflöst. Wird die Platine mit UV-Licht bestrahlt, während die transparente Folie mit dem Layout auf ihr liegt, wird der Fotolack während des Entwickelns an den lichtdurchlässigen Stellen zerstört. Daraufhin wird die Platine in ein Ätzbad mit Natriumpersulfat gegeben und die Kupferschicht löst sich genau dort auf. Anschließend werden die Löcher gebohrt, die Platine wird vom restlichen Fotolack mit einem Lösungsmittel gereinigt und es werden Kupferniete zur Durchkontaktierung vernietet. Nun werden die Bauteile aufgelötet und die Schaltung wird in Betrieb genommen.

Nach der Fertigstellung und Inbetriebnahme der Hardware wurde die *GUI* erstellt und der Empfang von Daten ermöglicht. Es wurden mehrere Kommandos, die per Funk geschickt werden können, realisiert, z.B. die Justierung des Motors und die Einstellung der Ist- und der Solltemperatur. Die Beschreibungen befinden sich im Kapitel 4.2.2 - Die Funkverbindung.

Im Anschluss daran arbeitete ich mich in das Konzept des *OSGi*-Frameworks ein. Dabei handelt es sich um ein Modulsystem für Java, bei dem Komponenten zur Laufzeit dynamisch hinzugefügt werden können. Genaueres dazu findet sich im Kapitel 4.3 - OSGi. Zur Programmierung wurde die Open Source-IDE *Eclipse* genutzt, weil sie bereits das *Equinox-OSGi* Framework beinhaltet und sie im Labor standardmäßig für die Java-Programmierung eingesetzt wird. So wurde eine softwareseitige Repräsentation der Heizungsregelungen erstellt, mit der man die Funkbefehle an diese schicken kann und welche außerdem noch eine automatische Adressvergabe beinhaltet.

Für die Anbindung von Sensoren und Aktoren an ein *OSGi* Framework im Rahmen des *AAL* existieren bereits Open-Source Middlewares, so z.B. *OpenAAL*/*UniversAAL*[2]. *OpenAAL* ist in Verbindung mit der Heizungssteuerung allerdings nicht sinnvoll einsetzbar, da im *OpenAAL* Konzept die komplette Logik in der Middleware steckt, die Aktoren/Sensoren werden nur ausgewertet/angesteuert. Im Falle

[1] [Online] http://www.cadsoft.de/eagle-pcb-design-software/product-overview/?language=de

[2] [Online] http://openaal.org/

der Heizungsregelung ist dies aber nicht gewollt, da sie auch ohne das *OSGi*-Framework voll einsatzfähig sein soll. Das Framework soll die Funktionalitäten nur erweitern. Dies verbessert auch die Stabilität des Systems, da durch die dezentrale Logik im Vergleich zur *OpenAAL*-Struktur bei einem Ausfall des *OSGi*-Frameworks die Heizungsregelung weiterhin die Temperatur regelt.

Die im Anschluss daran mit dem **S**tandard **W**idget **T**ool (*SWT*), einer in *Eclipse* integrierten Programmbibliothek zur Erstellung von Benutzeroberflächen, umgesetzte *GUI* stellt dem Nutzer eine einfache Bedienmöglichkeit für die Thermostate[3] zur Verfügung. So kann hier die aktuelle Temperatur ausgelesen, die Solltemperatur vorgegeben, ein Belüftungsmodus gestartet und ein Temperaturverlauf angezeigt werden. Zusätzlich kann man mit einem versteckten *Dialog* mit Entwickleroptionen auf weitere Funktionen zugreifen.

Im nächsten Kapitel wird die Beschreibung der Umsetzung mit der Hardware begonnen.

[3] Thermostat wird im Folgenden als Synonym für die Heizungsregelung verwendet.

4.1 Hardware

Die Hardware basiert größtenteils auf der im Praxisprojekt entwickelten Schaltung. Die Details sind in der dazugehörigen Dokumentation zu finden (Andreas, 2013). Gegenüber der Vorgängerversion geändert wurde die Position des Temperatursensors auf der Platine, er ist jetzt vom Mikrocontroller aus gesehen am anderen Ende der Platine, da dieser eine Temperaturdifferenz gegenüber der Raumtemperatur verursachen konnte. Weiterhin verursachte die Motoransteuerung während des Praxisprojekts Probleme, sodass eine Alternative gesucht wurde. Diese wird im Folgenden beschrieben.

4.1.1 Der Motortreiber

Um die Ansteuerung des Motors zu realisieren, wurde im Praxisprojekt eine H-Brücke eingesetzt. Diese Schaltung verdankt ihren Namen ihrem klassischen, in Abbildung 3 dargestellten, Aufbau, der einem „H“ ähnelt. Beim Einschalten eines jeweils diagonal gegenüber liegenden Transistorpaares dreht sich der Motor in die eine oder andere Richtung. Wichtig ist hierbei, dass auf keinen Fall zwei Transistoren auf einer Seite gleichzeitig eingeschaltet werden dürfen, da es sonst zu einem Kurzschluss kommt.

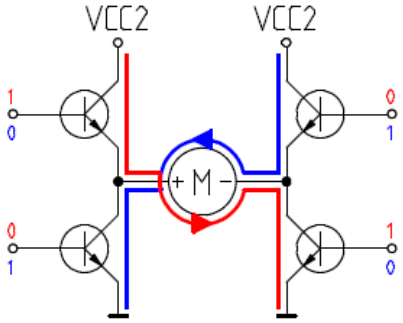


Abbildung 3: Prinzip der H-Brücke
(Gosch, 2006)

Die während des Praxisprojektes eingesetzte H-Brücke verursachte mehrfach Probleme (Andreas, 2013 S. 39), der Aufbau ist in Abbildung 4 zu sehen.

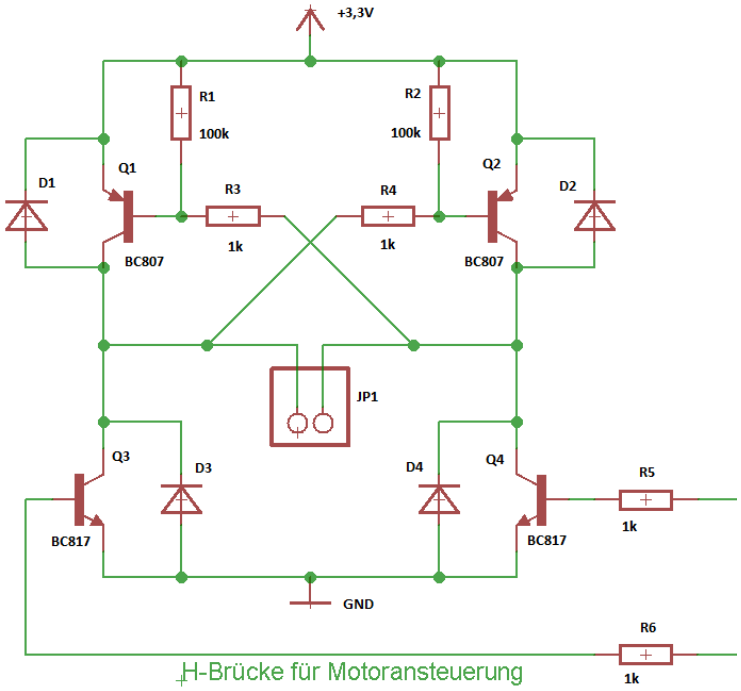
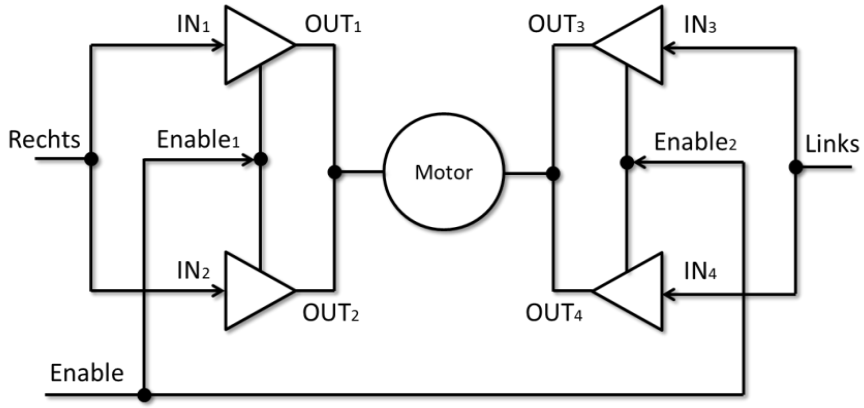


Abbildung 4: Schaltplan der H-Brücke

Das Basispotential der beiden PNP-Transistoren liegt nicht zwangsläufig auf Betriebsspannung, wenn z.B. der Transistor Q_1 schaltet, ist der Basiswiderstand R_4, für $U_{CESat} \approx 0{,}7\,V$, auf dem al $3{,}3V - U_{CESat} \approx 2{,}6V$. Dies kann, wenn z.B. ein Übergangswiderstand durch eine schlecht vernietete Kupferniete hinzukommt, den Transistor Q_2 ebenfalls schalten lassen, sodass es zu einem Kurzschluss kommt. Daher wurde entschieden, die H-Brücke entweder mit einem zusätzlichen Transistor für jede Seite zu versehen, der den PNP-Transistor beim Abschalten vom Massepotential trennt, oder einen fertigen Motortreiberbaustein zu verwenden.

Im Rahmen einer im Informatiklabor durchgeführten Bachelorthesis (Entwicklung einer mikrocontrollerbasierten, sensorgesteuerten Raumluftoptimierung in der Altenpflege, (Krüger, 2010)), wurde bereits ein fertiger Treiberbaustein eingesetzt, der *L293DD*. Dieser kann 4 Kanäle ansteuern, wobei jeder einen Ausgangstrom von bis zu 600mA liefern kann (SGS-THOMSON Microelectronics, 1996). Bei dem eingesetzten Motor, welcher aus einem fertigen Heizkörperthermostat von eQ-3 ausgebaut wurde, wurden aber nur maximal 180mA gemessen. Die Versorgungsspannung und die Ansteuerspannung der Logik können jeweils bis zu 36V betragen. Weil der Fokus der Bachelorthesis auf der Integrierung der Heizungsregelung in das Funknetzwerk des Informatiklabors liegt und ein in der Praxis erprobter Treiberbaustein keine großen Tests mehr benötigt, wurde entschieden, auf eine Weiterentwicklung der H-Brücke zu verzichten und stattdessen den *L293DD* zu verwenden.

Der Motortreiber befindet sich direkt an der Spannungsversorgung der Schaltung, da der Festspannungsregler nur 3,3V ausgibt und einen maximalen Strom von 150mA liefern kann. Bei der Inbetriebnahme der Schaltung sollten trotzdem keine 36V angelegt werden, da für den Motor kein Datenblatt vorhanden und die maximal zulässige Motorspannung unbekannt ist. 5V Versorgungsspannung reichen voll und ganz für den Betrieb aus. Praktischerweise enthält der Motortreiber schon Freilaufdioden an jedem Ausgang, sodass auch induktive Lasten, wie der Motor, geschaltet werden können, ohne dass andere Bauteile durch hohe Abschaltspannungen zerstört werden. Das Blockdiagramm inkl. Motor und die Schaltlogik des Bausteins sind in Abbildung 5 zu sehen.



TRUTH TABLE (one channel)

Input	Enable (*)	Output
H	H	H
L	H	L
H	L	Z
L	L	Z

Z = High output impedance
(*) Relative to the considered channel

Abbildung 5: Blockdiagramm und Schaltlogik Motortreiber
(SGS-THOMSON Microelectronics, 1996)

Um den Motor nun zu verfahren, muss die *Enable*-Leitung auf „*High*“ liegen und entweder „Rechts“ oder „Links“ auf „*High*“ und der andere auf „*Low*“.

Für den *L293DD* war keine *EAGLE*-Bibliothek vorhanden, sodass noch eine erstellt werden musste. Anschließend wurde die Hardware mit den Änderungen neu aufgebaut und in Betrieb genommen. Dies verlief problemlos, nur die Ruhestromaufnahme der Schaltung war recht hoch. Dies ist im Kapitel 5.2 - Motortreiber beschrieben. Die fertige Schaltung ist in Abbildung 6 zu sehen:

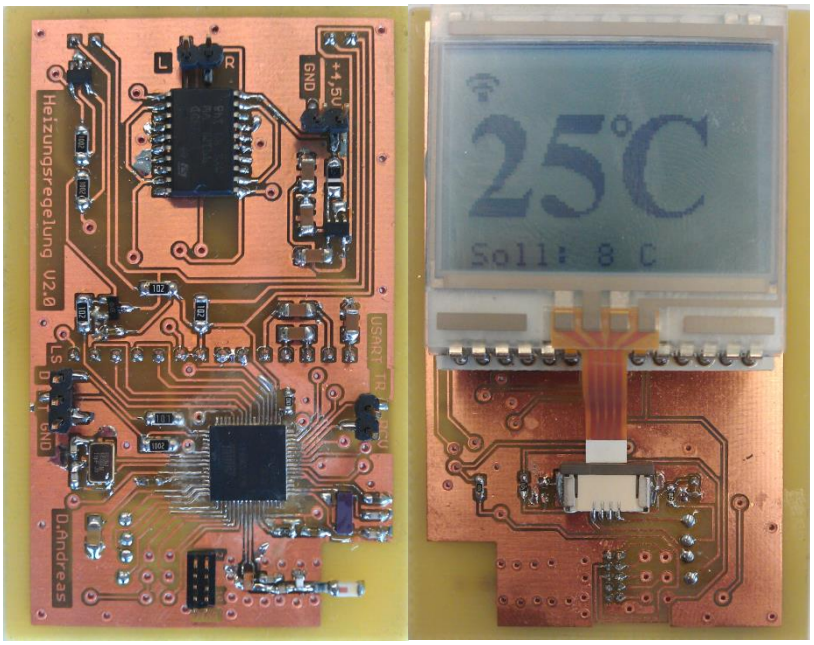


Abbildung 6: Fertige Heizungsregelung V2.0

Anschließend folgte die Erweiterung der Software.

4.2 Contiki

Bei der Bachelorthesis kam, wie auch bereits im Praxisprojekt, das von Adam Dunkels entwickelte Betriebssystem *Contiki* zum Einsatz. *Contiki* ist quelloffen und speziell auf Systeme mit geringer Speicherkapazität und geringem Energieverbrauch ausgelegt. Es ist ein multithreadingfähiges, eventgesteuertes System, bei dem die pausierten Thread auf Events warten, welche z.B. durch Timer oder andere Threads versandt werden. Eines der wichtigsten Features ist der ebenfalls von Adam Dunkels entwickelte *µIP-Stack*, ein *6LoWPAN*(IPv**6** over **Lo**w-power **W**ireless **P**ersonal **A**rea **N**etworks)-fähiger *TCP/IP*-Stack, welcher auch *ICMP* und *UDP* unterstützt. Eine genauere Beschreibung von *Contiki* findet sich in meinem Praxisprojekt (Andreas, 2013 S. 26).

Die einzelnen Threads haben in *Contiki* keinen eigenen *Stack*[4], weshalb sie von Adam Dunkels „*Protothreads*" getauft wurden. Der Entwickler muss selber dafür Sorge tragen, dass alle kritischen Variablen vor einem Pausieren des Threads gesichert werden oder dass sie mit dem Schlüsselwort *static* versehen sind. Dadurch legt der Compiler Variablen mit einer festen Speicheradresse an, wobei der Speicherbereich nicht überschrieben wird. Der zugrunde liegende Gedanke der *Protothreads* ist der daraus resultierende geringe Speicherbedarf.

Üblicherweise wird ein *Protothread* gestartet, führt eine Initialisierung aus und wartet danach auf ein für ihn bestimmtes Event. Danach führt er seine eigentliche Funktion aus und wartet auf das nächste Event. Wichtig ist hierbei, dass die Threads nicht aktiv warten. Der Kernel reiht alle asynchron verschickten Events in die *Eventqueue* ein, ruft den jeweiligen Ziel-*Protothread* des Events auf und übergibt ihm dieses als Parameter. Events können auf zwei verschiedenen Arten versandt werden. Der Befehl process_post() sendet ein asynchrones Event, welches in die *Eventqueue* eingereiht wird, der Befehl process_post_synch() umgeht diese und ruft den Ziel-*Protothread* sofort auf.[5] Im Laufe der Bachelorthesis wurde entdeckt, dass die *Eventqueue* 32 Events fasst, dies ist leider in keiner der durchsuchten Dokumentationen festgehalten. Alle weiteren Events, die man zu verschicken versucht, produzieren zwar keine Fehlermeldung oder einen Neustart des Systems, kommen aber auch nie beim Ziel an.

Während des Praxisprojektes kam es vor, das einzelne *Protothreads* bei *Contiki* „einfroren". Um dies zu beheben, wurden die Debuggingmöglichkeiten von *Contiki* genutzt um den Fehler zu finden, dies ist im Kapitel 5.1 - Contiki-Events beschrieben.

Nach der Behebung dieses Problems wurde eine Benutzeroberfläche aufgebaut, mit der die Vorgabe des Temperatur-Sollwerts über den Touch Panel des LC-Displays möglich ist. Alle weiteren Funktionen wurden wegen der begrenzten Darstellungsfläche und der damit schlechten Bedienbarkeit später in die *OSGi*-Anwendung ausgelagert. Die Beschreibung der Erstellung der *GUI* folgt im nächsten Kapitel: 4.2.1 - Die grafische Benutzeroberfläche. Anschließend wird die Umsetzung der Funkverbindung beschrieben. Die Funkbefehle wurden parallel zur Erstellung der *OSGi*-Anwendung implementiert, welche auf einem PC läuft und mit der das Thermostat gesteuert werden kann. Die *OSGi*-Anwendung wird im Kapitel 4.3 - OSGi beschrieben. Die Programmablaufpläne für die selbsterstellten *Protothreads* befinden sich auf der CD im Anhang.

[4] Stack = Stapelspeicher, Speicherbereich im Arbeitsspeicher der den aktuellen Threadzustand enthält

[5] Eine gute Übersicht über die *Protothread*-Verwaltung findet sich online auf https://github.com/contiki-os/contiki/wiki/Processes. (Stand 24.07.2013)

4.2.1 Die grafische Benutzeroberfläche

Zum Bedienen und Beobachten der Heizungsregelung wird, wie im Kapitel 2-Bestandsaufnahme beschrieben, ein LC-Display mit einer weißen LED-Hintergrundbeleuchtung und ein resistives Touch-Panel, beide hergestellt von Electronic Assembly, verwendet. Die Darstellungsfläche des Displays beträgt 102 x 64 Pixel, wobei die Spalten jeweils in Seiten von jeweils 8 Pixeln organisiert sind. Dies ist in Abbildung 7 dargestellt.

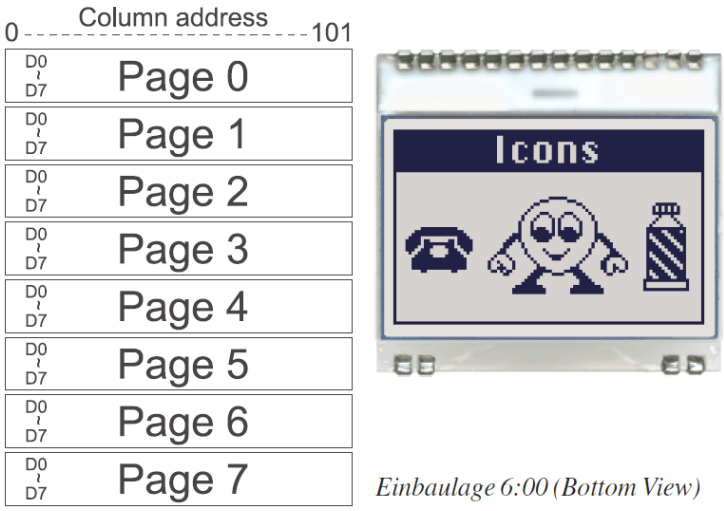


Abbildung 7: Aufbau der Displaydarstellung (ELECTRONIC ASSEMBLY GmbH, 2011)

Die Ansteuerung des Displays erfolgt über *SPI*, welcher vom *ATmega128RFA1* unterstützt wird. *SPI* (Serial Peripheral Interface) ist ein synchroner, serieller, vollduplexfähiger Master-Slave-Datenbus, welcher von Motorola entwickelt wird. Die bereits im Praxisprojekt benutzten Treiberroutinen für *SPI* und für das Display werden auch weiterhin verwendet (Andreas, 2013 S. 5). Zum Verständnis der weiteren Nutzung der Treiber werden diese trotzdem kurz erläutert. Dazu müssen die Dateien *Display_DOGS102.c* und *spi.c* mit den dazugehörigen Headerdateien in das Projekt implementiert werden. Hier müssen nun einige Einstellungen vorgenommen werden, wie die passenden Ports für das Einschalten der LED-Hintergrundbeleuchtung, den *Display-Reset* und den *Chip-Select*. Der Treiber beinhaltet einige nützliche Funktionen um das Display anzusteuern, welche anhand eines Beispiels kurz gezeigt werden:

```
1  init_SPI();
2  init_DISPLAY();
3  display_clear();
4  show_TEXT(0,0,"Initialisiere",0,0);
5  display_data(1,0,0b11001100);
```

Listing 1: *Display*-Treiber Beispiel

Mit dem Befehl `init_SPI();` wird in Zeile 1 die *SPI*-Schnittstelle initialisiert. Dazu werden die Bits im *SPI*-Kontrollregister des Controllers für die Übertragungsgeschwindigkeit, die Orientierung der Daten, den Master-Modus und die Takt-Polarität gesetzt. Um das Display zu verwenden wird der Befehl `init_DISPLAY();` aufgerufen, welcher eine Reihe von Konfigurationsbytes an das dieses schickt, z.B. für den Kontrast, die Displayinvertierung, usw. Eine genaue Auflistung befindet sich im Datenblatt (ELECTRONIC ASSEMBLY GmbH, 2011). Der Befehl `show_TEXT(0,0,"Initialisiere",0,0);` stellt beginnend mit Seite Null, Spalte Null den Text „Initialisiere“ dar. Durch den Aufruf von

`display_data(1,0,0b11001100);` wird bei Seite Eins, Spalte Null das übergebene Byte auf dem Display gezeigt, wobei die Pixel, die mit einer 1 angesteuert werden, dunkel werden.

Das Display beinhaltet keinen vorkonfigurierten Zeichensatz, jede Darstellung muss durch die Übergabe von der Seite, der Spalte und dem zu schreibenden Byte aufgebaut werden. Die Treiberroutine `show_TEXT` beinhaltet den für die Darstellungsgröße einer Seite = 8 Pixel hinterlegten Zeichensatz. Für die Erstellung einer grafischen Benutzeroberfläche, die auch von älteren Menschen benutzt werden soll, ist diese Größe allerdings nicht ausreichend. Deshalb mussten neue Zeichen erstellt werden. Zur Darstellung der Temperatur sind die Zahlen von 0-9, sowie das „°C“-Zeichen und ein Minuszeichen für negative Temperaturen nötig. Außerdem wurden noch zwei Pfeile realisiert, um die Erhöhung bzw. Verringerung der Solltemperatur optisch darzustellen. Die Zahlen von 0-9 wurden in einer Höhe von 4 Seiten und einer Breite von 24 Zeilen angelegt, wobei ein Zeichenprogramm mit der Schriftart „Times New Roman“ mit einem hinterlegten Raster verwendet wurde, um die Pixel einfacher zu zählen. Jedes Byte wurde anschließend in ein Array geschrieben. Die fertige grafische Benutzeroberfläche ist in Abbildung 8 dargestellt.

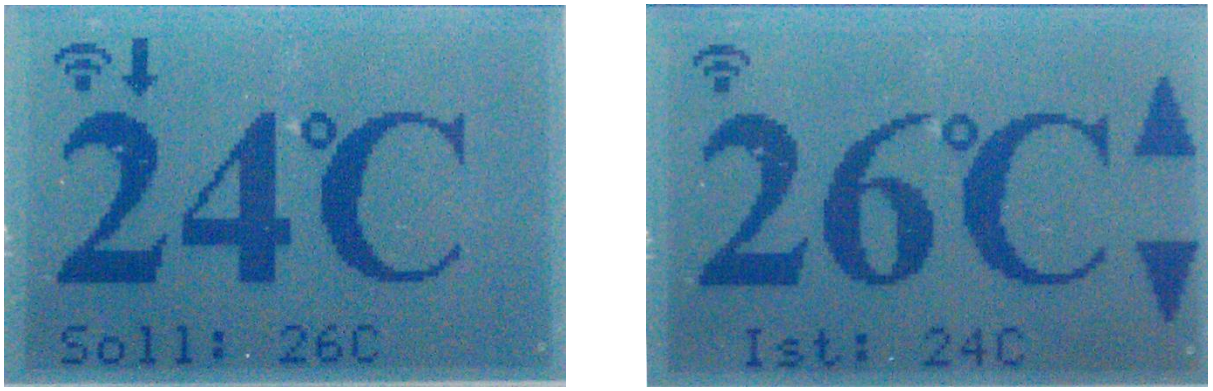


Abbildung 8: Grafische Benutzeroberfläche

Es existieren zwei Benutzeroberflächen, zwischen denen der Nutzer mit einem Druck auf das Touch Panel wechseln kann. Standardmäßig wird auf dem Bildschirm groß die aktuell gemessene Temperatur, und klein am unteren Rand die Solltemperatur gezeigt. Diese Oberfläche ist in Abbildung 8 auf der linken Seite zu sehen. Nach dem Druck auf das Touch Panel wird die rechts sichtbare Oberfläche angezeigt und die Displayhintergrundbeleuchtung wird eingeschaltet. Mit einem Druck auf die obere Displayhälfte wird die Solltemperatur, die jetzt groß sichtbar ist, erhöht, mit einem Druck auf die untere verringert. Der Nutzer muss den Druck dabei aufrechterhalten, da die Änderung der Temperatur nur alle 0,5 Sekunden erfolgt. Ohne die Wartezeit würde die Temperatur innerhalb von Millisekunden auf dem Endwert sein. Wenn kein Druck mehr festgestellt wird, wird nach fünf Sekunden die Anzeige wieder auf die aktuelle Temperatur umgeschaltet und nach weiteren fünf Sekunden geht die Displaybeleuchtung aus.

Bei einer Höhe von 4 Seiten und einer Breite von 24 Zeilen ergibt sich bei zehn Ziffern eine Datenmenge von 960 Byte. Mit den Pfeilen, dem „°C“-und dem Minus-Zeichen sind es insgesamt 1280 Bytes. Diese Datenmenge füllt den Arbeitsspeicher des Mikrocontrollers unnötigerweise und weil die Daten nicht geändert werden müssen wurden sie in das *EEPROM* verlagert. Die Arrays wurden in einer eigenen Headerdatei hinterlegt, *temperature_characters.h*. In dieser wird zusätzlich mit der Zeile `#include <avr/EEPROM.h>` die AVR-Headerdatei für die Nutzung des *EEPROMS* eingebunden. Die Speicherung der Daten im *EEPROM* erfolgt anschließend durch den Compiler mit dem Schlüsselwort *EEMEM*:

```
unsigned char arrow_up[2][24] EEMEM = {{0b00100100,0b…},{…}};
```

Für den Zugriff auf die Daten bietet der *EEPROM* Treiber verschiedene Funktionen für verschiedene Datenlängen an, z.B. `EEPROM_read_byte` oder `EEPROM_read_block`. Diesen muss nur der Variablenname übergeben werden, sowie im Falle von `read_block` eine Byteanzahl.

Dadurch kann Platz im Arbeitsspeicher gespart werden, da während der Erstellung der *GUI* festgestellt wurde, dass der Arbeitsspeicher des Controllers nicht ausreichte und der *Contiki-Protothread* für das Versenden von Debugginginformationen über die *USART*-Schnittstelle abstürzte. Darauf wird im Kapitel 5.3 - EEPROM nochmals eingegangen.

Um die Funkverbindung des Thermostats zu visualisieren, wurde zusätzlich noch ein Symbol erstellt, welches angezeigt wird, sobald Daten versandt oder empfangen wurden. Dieses ist in Abbildung 9 zu sehen. Zwei kleine Pfeile direkt daneben zeigen die Datenrichtung an, wobei der nach oben zeigende Pfeil für versandte und der nach unten zeigende für empfangene Daten steht. Dazu wird von dem *Protothread*, der für die Funkkommunikation zuständig ist, ein Event an den *GUI-Protothread* versandt. Jeweils drei Sekunden nach dem Versenden/Empfangen werden die Pfeile wieder ausgeblendet. Die Umsetzung der Funkverbindung wird im nächsten Kapitel 4.2.2 - Die Funkverbindung beschrieben.

Abbildung 9: *GUI*-Darstellung der Funkverbindung

Ein weiteres realisiertes Symbol wird am oberen Displayrand angezeigt, sobald der Belüftungsmodus aktiviert wurde. Dafür wurde ein Unterprogramm geschrieben, welches vom *Main-Protothread* entsprechend aufgerufen wird. Das Symbol ist in Abbildung 10 dargestellt.

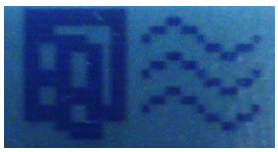


Abbildung 10: *GUI*-Darstellung des Belüftungsmodus

Für die Darstellung wurden verschiedene Unterprogramme geschrieben, um die Übersichtlichkeit und Wiederverwendbarkeit des Codes zu erhöhen.

- ```
  void show_arrow(unsigned char page, unsigned char column,
  unsigned char direction);
  ```

  Diese Funktion stellt einen Pfeil von zwei Seiten Höhe und 24 Pixeln Breite auf dem Display dar, wobei die Seite und die Zeile, an der die Darstellung aufgebaut werden soll, übergeben wird. Diese entsprechen der oberen, linken Ecke der Darstellung. Mit dem Parameter direction wird die Richtung des Pfeils eingestellt, bei Übergabe von `ARROW_UP` zeigt er nach oben, bei `ARROW_DOWN` nach unten.

  Die Pfeile dienen dem Hinweis für den Nutzer, dass er die Solltemperatur mit einem Druck auf die obere Displayhälfte erhöhen und mit einem Druck auf die untere verringern kann.

- ```
  void display_temperature_character (unsigned char page, unsigned char column,
  char number);
  ```

Die Funktion *display_temperature_character (…)* bekommt als Übergabewert ebenfalls ihre jeweiligen Startpunkte, die wieder in der linken oberen Ecke liegen, sowie ein ASCII-Zeichen im char-Format. Der Zeichensatz ist im *EEPROM* hinterlegt und umfasst die Zahlen von 0-9, das Minuszeichen, das Leerzeichen und bei der Übergabe des Buchstaben „C“ wird „°C“ ausgegeben. Diese Funktion stellt die großen Zeichen für die Temperaturanzeige zur Verfügung.

- `void show_temperature (unsigned char page, unsigned char column, char temperature);`

  void show_temperature (…) benötigt ebenfalls die Startseite und Startzeile und eine im Datentyp char übergebene Dezimalzahl zwischen -99 und 99. Ist die Zahl betragsmäßig größer als 99, wird sie auf 99 gesetzt. Sie wird anschließend auf dem Display ausgegeben, wobei automatisch auch das „°C“-Zeichen angefügt wird. Im Endeffekt bereitet sie die Funktion *display_temperature_character(…)* weiter auf.

- `void show_data_direction(unsigned char page, unsigned char column, char direction);`

  Um die Funkverbindung der Heizungsregelung zu visualisieren wurde diese Funktion erstellt. Sie stellt an den übergebenen Startpunkten ein Funksymbol dar, wobei rechts daneben zwei Pfeile in Abhängigkeit der übergebenen *direction* dargestellt werden. Folgende Möglichkeiten gibt es:

  - *RECEIVE* – Ein Pfeil nach unten wird gezeigt.
  - *RECEIVE_NONE* – Der Pfeil nach unten wird gelöscht.
  - *SEND* – Ein Pfeil nach oben wird gezeigt.
  - *SEND_NONE* – Der Pfeil nach oben wird gelöscht.

  Das Löschen der Datenrichtungspfeile wird im *GUI-Protothread* automatisch nach drei Sekunden durchgeführt.

- `void show_airing_symbol(char Display_Flag);`

  Damit der Nutzer eine Rückmeldung bekommt, wenn der Belüftungsmodus eingeschaltet wurde, kann mit dieser Funktion ein Belüftungssymbol auf dem Display angezeigt werden. Bei Übergabe einer Null wird das Symbol gelöscht, bei einem anderen Wert angezeigt.

- `void show_actual_screen(void);` & `void show_reference_screen(void);`

  Der Übersichtlichkeit und Wiederverwendbarkeit wegen wurde die Darstellung der beiden möglichen Benutzeroberflächen in zwei Unterfunktionen gepackt, *void show_reference_screen(void)* für die Eingabe der Solltemperatur und *void show_actual_screen(void)* für den gemessenen Wert. Die Programmablaufpläne befinden sich auf der beigefügten CD.

4.2.2 Die Funkverbindung

In diesem Kapitel wird auf die softwareseitige Realisierung der Funkanbindung eingegangen. Die Hardware, die nötig ist um diese Verbindung herzustellen, besteht aus einer 2,4Ghz Keramik-Antenne von Johanson Technology und einem im *ATmega128RFA1* vorhandenen Funkchip. Auf die Details wird hier nicht mehr eingegangen, da diese Teil des Praxisprojekts waren und in der Dokumentation nachgelesen werden können (Andreas, 2013 S. 17).

Im Praxisprojekt wurde die Funkanbindung nicht mehr realisiert, die gemessene Temperatur wurde lediglich per *6LoWPAN/UDP-Broadcast* versandt. Dazu wurde der *µIP-Stack* verwendet, der in *Contiki* integriert ist.

4.2.2.1 6LoWPAN

6LoWPAN ist ein Kommunikationsprotokoll, welches die neueste Version des Internetprotokolls(*IPv6*) für energieeffiziente Systeme umsetzt. *IPv4* wird nicht unterstützt. Die *LoWPAN* - Anpassungsschicht ermöglicht es, *IPv6* Pakete über „Low-Rate Wireless Personal Area Networks“ (IEEE 802.15.4) zu versenden (Shelby, et al., 2009 S. 16). Die Anpassungsschicht ist in Abbildung 11 zu sehen.

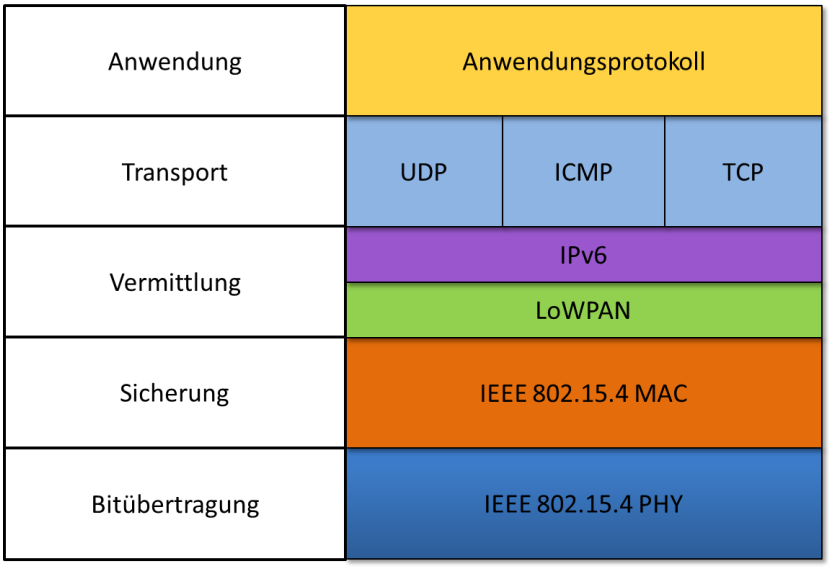


Abbildung 11: *LoWPAN*-Anpassungsschicht im *OSI*-Modell (Shelby, et al., 2009 S. 16)

4.2.2.2 Senden von Daten

Der *µIP-Stack* in *Contiki* setzt diese Spezifikationen bereits um. Für das Versenden und Empfangen von *UDP*-Daten gibt es seit *Contiki* V2.5 die *API* „*Simple UDP*“. Ein kurzes Codebeispiel verdeutlicht die Nutzung dieser Schnittstelle:

```
1  static struct simple_udp_connection broadcast_connection;
2  static uip_ipaddr_t addr;

3  PROCESS(broadcast_process, "UDP Broadcast Process");
4  PROCESS_THREAD(broadcast_process, ev, data)
5  {
6  PROCESS_BEGIN();

7  uip_create_linklocal_allnodes_mcast(&addr);

8  simple_udp_register(&broadcast_connection, UDP_PORT,NULL, UDP_PORT,receiver);
   //UDP-Verbindung registrieren

9  simple_udp_sendto(&broadcast_connection, "Hello World!",
10 (strlen("Hello World!")), &addr);

11 PROCESS_END();
12 }
```

```
13 static void receiver(struct simple_udp_connection *c,
14 const uip_ipaddr_t *sender_addr,
15 uint16_t sender_port,
16 const uip_ipaddr_t *receiver_addr,
17 uint16_t receiver_port,
18 const uint8_t *uip_data,
19 uint16_t datalen)
20 {
21 }
```

Listing 2: UDP-Connection Beispiel

Zuerst werden zwei statische Variablen angelegt, in Zeile 1 `static struct simple_UDP_connection broadcast_connection;` wodurch eine Struktur erstellt wird, die alle relevanten Daten, die *Contiki* für das Versenden der Daten braucht, enthält und in Zeile 2 wird eine *IP*-Adresse für den *µIP-Stack* deklariert. Zeile 3 macht anschließend dem System den *Protothread* „broadcast_process“ bekannt.

Die Definition des *Protothreads* folgt ab Zeile 4, wobei der erste folgende Befehl PROCESS_BEGIN(); in jedem *Protothread* vorhanden sein muss, genau wie PROCESS_END(); am Ende.

Mit dem Befehl `uip_create_linklocal_allnodes_mcast(&addr);` in Zeile 7 wird eine *Multicast IPv6* Adresse in der Variablen *addr* gespeichert.

Die *UDP*-Verbindung muss dem System anschließend mit `simple_UDP_register(&broadcast_connection, UDP_PORT,NULL, UDP_PORT,receiver);` bekannt gemacht werden. *broadcast_connection* ist hierbei die in Zeile 1 deklarierte *simple_UDP_connection*, *UDP_PORT* eine Konstante die den gewünschten *UDP*-Port, auf dem gesendet werden soll, beinhaltet (im Informatiklabor z.B. 4321). Anstelle von *NULL* kann man eine *IPv6*-Adresse übergeben, woraufhin die Verbindung nur auf Daten dieser Adresse reagiert. *NULL* sorgt dafür, dass jegliche Sender, die auf dem folgenden *UDP_PORT* senden, akzeptiert werden. Der letzte Parameter ist die Funktion, die aufgerufen wird, wenn Daten empfangen wurden, in diesem Fall die in den Zeilen 13-21 definierte Funktion *receiver*.

Mit `simple_UDP_sendto(&broadcast_connection, "Hello World!",(strlen("Hello World!")), &addr);` wird schließlich mit der *broadcast_connection* der String „Hello World!“ an die zuvor registrierte *Multicast*-Adresse versandt, die Menge der Daten wird mit `strlen("Hello World!")` ebenfalls übergeben.

Die *receiver*-Funktion bekommt die in Zeile 13-20 beschriebenen Übergabeparameter vom System überreicht. Der Aufruf der Funktion erfolgt letztendlich asynchron durch das vom *µIP-Stack* generierte *ip_event*, wobei die übergebenen Daten in der *Simple UDP API* als *static* deklariert sind. In der Funktion kann man dann die Daten auslesen und entsprechend reagieren.

Der während der Bachelorthesis realisierte *Protothread* „*broadcast_process*“ wartet nach dem Starten und der Registrierung der *UDP*-Verbindung auf das Event *e_UDP_send*. Wird dieses empfangen, werden die im Event angehängten Daten automatisch mit dem im Labor üblichen Protokoll unverschlüsselt als *Multicast* versandt. Außerdem wird das Event *e_UDP_receive* mit dem Inhalt „*send*“ an den *GUI-Protothread* versandt. Dieser stellt daraufhin für drei Sekunden ein kleines nach oben zeigendes Pfeilchen neben dem Funkverbindungssymbol dar. Der Programmablaufplan befindet sich auf der CD im Anhang.

Das Laborprotokoll für den Nachrichteninhalt ist wie in Abbildung 12 zu sehen aufgebaut:

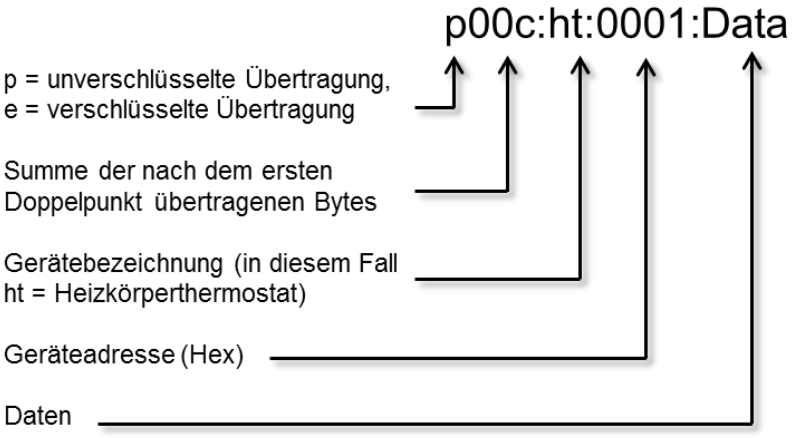


Abbildung 12: Aufbau des Nachrichteninhalts im Laborprotokoll

Die Geräteadresse ist dabei im *EEPROM* des Controllers hinterlegt und bleibt somit auch nach einem Ausfall der Stromversorgung erhalten. Die Adresse kann auch mit einem Funkbefehl geändert werden, in diesem Fall wird die neue Adresse im *EEPROM* gespeichert.

4.2.2.3 Empfangen von Daten

In der Receiver-Funktion, welche in Listing 2 in den Zeilen 13-21 beispielhaft definiert ist, können die auf *UDP*-Port 4321 eingetroffenen Daten ausgelesen und entsprechend reagiert werden.

In der während der Bachelorthesis umgesetzten Empfängerfunktion muss jeder Befehl an die Heizungsregelung, bis auf den Identifikationsbefehl *ht_id_request*, die Gerätebezeichnung und -adresse beinhalten, ansonsten wird nicht reagiert. Das Schema ist hierbei Gerätebezeichnung:Geräteadresse:Befehl, also z.B. „*ht:0001:adjust*“, um die Justierung des Motors des Thermostats mit der Adresse 0001 zu starten. Folgende Befehle sind möglich:

- *ref_temp_XX:*
  Stellt die Zieltemperatur ein, auf die die Heizungsregelung regeln soll. XX stellt hierbei die Temperatur dar. Die Temperatur muss zwischen 8°C und 25°C liegen, ansonsten wird sie verworfen. 8°C als Mindesttemperatur dienen dem Frostschutz und 25°C als Obergrenze sollen verhindern, dass versehentlich eine zu große Temperatur eingestellt wird.

- *pc_act_temp_:*
  Stellt die „gemessene“ Temperatur des Thermostats ein. Wird genutzt, um die vom Raumsensor gemessene Temperatur an das Thermostat zu übertragen und diese zu verwenden, als hätte die Heizungsregelung diese selbst gemessen. Wenn zwei Minuten lang keine externe Temperatur empfangen wird, verwendet die Heizungsregelung den eigenen Temperatursensor.

- *adjust:*
  Startet die Justierung des Motors.

- *change_ID XXXX:*
  Ändert die Geräteadresse auf die übergebene vierstellige Hexadezimalzahl. Anschließend wird eine Bestätigung mit der alten und der neuen Adresse verschickt.

- *ht_id_request:*
  Auf diesen Befehl wird auch reagiert, wenn er nicht an das eigene Thermostat gerichtet ist. Beim

Empfang wird als Antwort „*ht_id_ack*“ gesendet, nachdem eine zufällige Zeit zwischen 1 und 100ms gewartet wurde. Die Wartezeit soll verhindern, dass bei mehreren Thermostaten im Netzwerk eine der Antworten nicht vom Empfänger erkannt wird. Er dient der Anmeldung der Thermostate im Netzwerk.

- *airing_mode:*
  Startet den Belüftungsmodus. Die Regelung wird zehn Minuten lang nicht aufgerufen und der Motor fährt den Ventilstift aus. Außerdem wird das Belüftungssymbol auf dem Display angezeigt, solange der Modus aktiv ist.

Werden Daten empfangen wird immer das *e_UDP_receive* Event mit dem Datum „*receive*“ an den *GUI-Protothread* geschickt, welcher daraufhin drei Sekunden lang ein nach unten zeigendes Pfeilchen neben dem Funkverbindungssymbol darstellt. Alle empfangenen *6LoWPAN*-Daten werden, unabhängig davon, ob sie für das Gerät bestimmt sind, im Debugging-Modus über *USART* ausgegeben. Der Programmablaufplan für die Empfängerfunktion befindet sich auf der CD im Anhang.

Damit ist die Umsetzung der Funkanbindung und der grafischen Benutzeroberfläche abgeschlossen. Im nächsten Kapitel wird zuerst das *OSGi*-Konzept vorgestellt und danach die Umsetzung der Software beschrieben.

4.3 OSGi

Die zentrale Verwaltungskomponente für viele der im *WieDAS* Forschungsprojekt realisierten Geräte bildet ein PC mit einem darauf laufenden *OSGi*-Framework. In diesem Kapitel wird zunächst kurz auf das Konzept der *OSGi Service Plattform* eingegangen und danach die praktische Umsetzung erläutert.

OSGi (früher: **O**pen **S**ervices **G**ateway **i**nitiative, inzwischen ohne Bedeutung) ist ein Modulsystem für Java, bei dem Komponenten zur Laufzeit dynamisch hinzugefügt werden können. Spezifiziert wird *OSGi* von der *OSGi Alliance*, welche ein Zusammenschluss von Unternehmen wie Ericsson, IBM und Oracle ist, der ursprünglich 1999 gegründet wurde (Wütherich, et al., 2008 S. 12). Dabei gibt die *OSGi Alliance* allerdings nur die Spezifikationen vor, die ein *OSGi*-Framework erfüllen muss. Die technische Umsetzung obliegt dem jeweiligen Hersteller.

Die Softwarekomponenten oder Module, die in das Framework integriert werden können, werden *Bundles* genannt, welche von außen zugängliche *Services* (Dienste) zur Verfügung stellen können.

„Ein Bundle ist eine fachlich oder technisch zusammenhängende Einheit von Klassen und Ressourcen, die eigenständig im Framework installiert und deinstalliert werden kann."

(Wütherich, et al., 2008 S. 12)

Durch diese Unterteilung von zusammenhängenden Softwareeinheiten in *Bundles* kann eine hohe Modularisierung erreicht werden, bei der *Bundles*, die keine Abhängigkeiten voneinander haben, zu völlig unterschiedlichen Zeitpunkten während der Laufzeit des Systems installiert werden können.

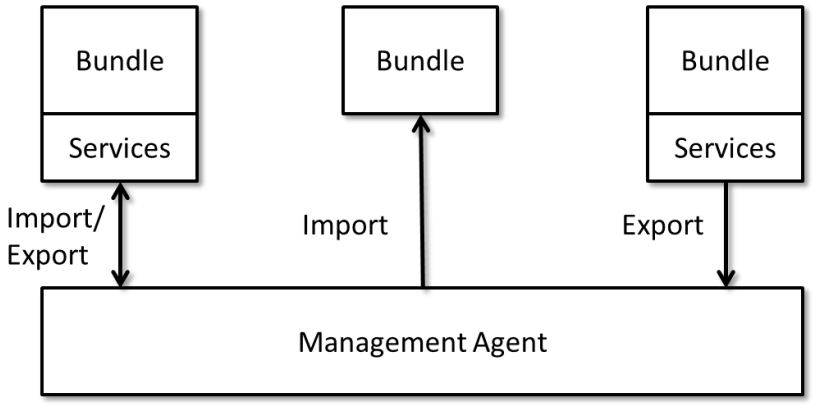


Abbildung 13: *Bundle* & *Services* Abhängigkeiten

Bundles, die auf *Services* von anderen *Bundles* zugreifen wollen, müssen diese zuvor importieren. Sollen andere *Bundles* auf ihre *Services* zugreifen können, so müssen diese exportiert werden. Dieses Prinzip ist in Abbildung 13 zu sehen. *OSGi*-Frameworks bringen zusätzlich einige *Services* mit, wie z.B. den *Event Admin Service*, mit dem Events zwischen den *Bundles* verschickt werden können.

Externe *Bundles* können entweder als *required* eingestellt werden oder als *imported*. Der Unterschied ist, dass bei einem *required Bundle* im Gegensatz zu einem *imported Package* sämtliche Abhängigkeiten desselben mit übernommen werden und dass das *Bundle*, das darauf zugreifen möchte, bei einem Fehler des referenzierten *Bundles* ebenfalls nicht lauffähig ist. Im Allgemeinen wird deshalb empfohlen *imported Package*s zu verwenden. (Kriens, 2008)

Es existieren verschiedene *OSGi*-Frameworks, z.B. *Apache Felix*, *Knopflerfish* oder *Equinox*. Die Open-Source-IDE *Eclipse* basiert auf *Equinox* und wird im Informatik Labor für die Java-Programmierung eingesetzt, weshalb auch das *Equinox*-Framework verwendet wird. *Eclipse* wird

durch *Equinox* als Basis das Installieren von Komponenten zur Laufzeit ermöglicht und es können verschiedene Plug-Ins, wie z.B. für die C++- oder *Android*- Programmierung nachinstalliert werden. *Eclipse* wurde während der Bachelorarbeit ebenfalls für die Java-Programmierung eingesetzt.

Um die verschiedenen Geräte des *WieDAS*-Forschungsprojektes mit *OSGi* zu verwalten, wird im Informatiklabor die in Abbildung 14 zu sehende Struktur eingesetzt

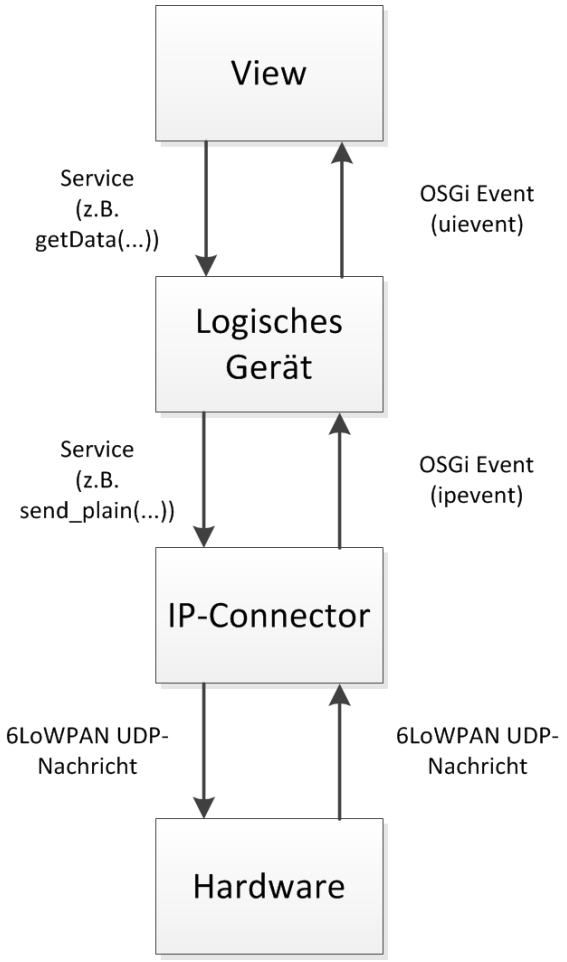


Abbildung 14: *OSGi-Bundlestruktur* des Informatik Labors

Von oben nach unten werden dabei *Services* aufgerufen, welche die darunter liegenden *Bundles* zur Verfügung stellen, von unten nach oben werden Events versandt. Weil eine Baumstruktur vorliegt, bei der z.B. viele *Bundles* auf das *IP-Connector-Bundle*[6] zugreifen wollen, um die über *6LoWPAN* von den *WieDAS*-Geräten verschickten Daten auswerten zu können, muss dieses nicht die *Services* sämtlicher darüber liegender *Bundles* kennen. Die Implementierung von neuen *Bundles* in einer höheren Ebene erfordert somit keine Änderungen an den darunter liegenden Schichten.

Die unterste Ebene bildet dabei das physikalische Gerät, in diesem Fall die Heizungsregelung. Sämtliche Funkkommunikation mit ihr wird über das *IP-Connector Bundle* abgewickelt. Dieses schickt beim Empfang von Daten ein *ipevent* an sämtliche darüber liegenden *Devices* (über den *EventAdmin Service* von *OSGi*). Dies sind die logischen Geräte, welche eine Softwarerepräsentation der jeweiligen physikalischen Geräte sind. Darüber liegt die grafische Benutzeroberfläche, welche einen *Eventhandler* für

[6] Das *IP-Connector Bundle* wurde von Mirco Kern, einem Mitarbeiter des Informatik Labors, erstellt und ermöglicht das Versenden und Empfangen von *UDP-Broadcasts*, entweder verschlüsselt oder als Klartext.

das *uievent* implementiert, welches von allen logischen Geräten versandt werden kann. Im Gegenzug stellen die Geräte *Services* zur Verfügung, mit denen die *Views* mit dem tatsächlichen Gerät kommunizieren können. Die *Services* des *Devices* wiederrum greifen auf *Services* des *IP-Connectors* zurück, um Daten zu versenden. Diese Struktur ermöglicht eine sehr gute Modularisierung, sodass z.B. auch mehrere unabhängige Benutzeroberflächen mit ein- und demselben Gerät kommunizieren können. Die praktische Umsetzung der Einbindung der Heizungsregelung wird im Folgenden behandelt, wobei zuerst das logische Gerät (das *Device*) und danach die Benutzeroberfläche (die *View*) beschrieben werden. Die Klassendiagramme wurden mit dem kostenlosen *Eclipse* Plug-In *ObjectAid UML Explorer*[7] erstellt, das vollständige Klassendiagramm befindet sich auf der CD im Anhang

4.3.1 Das Device

Das logische *Device* ist eine softwareseitige Repräsentation der Heizungsregelung und stellt alle Funktionen für die Kommunikationen mit dieser als *Services* zur Verfügung. Zusätzlich ist eine Datenbank im *Device* eingebunden, in der alle Eigenschaften wie der Name des zugehörigen Raums, die Adresse, die gemessenen Temperaturen mit Zeitstempel usw. gespeichert werden. In der folgenden Abbildung 15 ist die Projektstruktur des Heizungsregelungs-*Device* zu sehen.

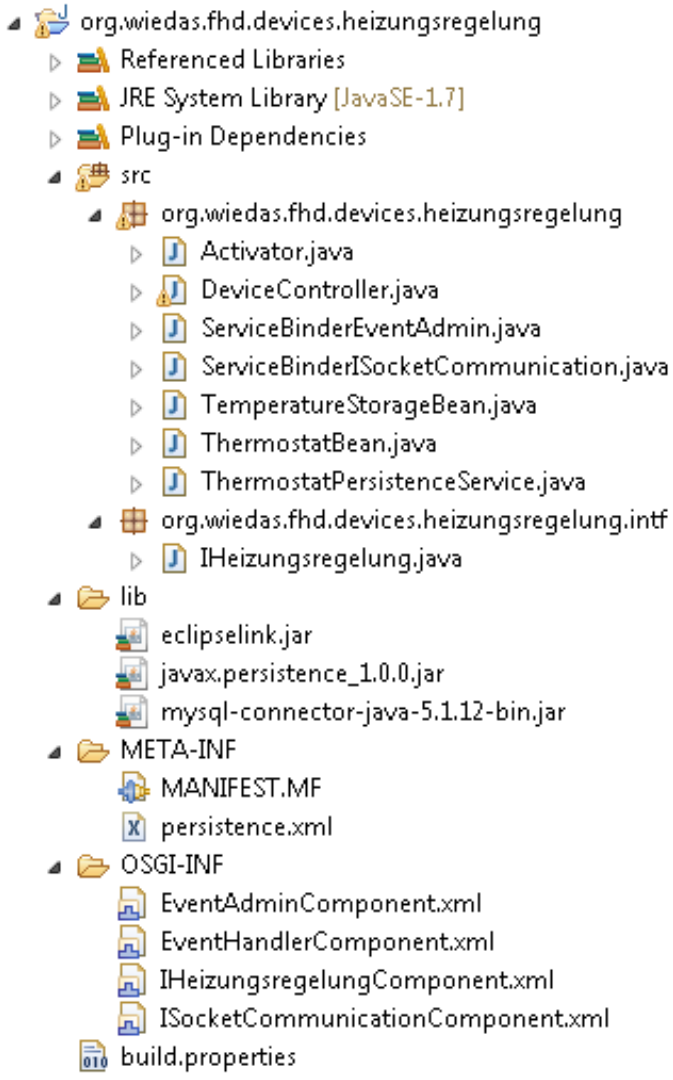


Abbildung 15: *Device* Projektstruktur

[7] http://www.objectaid.com/home

Eine der wichtigsten Dateien ist die *Manifest*-Datei, in der die *Metadaten*[8] gespeichert werden. So sind z.B. die *Components*, die *ServiceBinder* und die *Activator*-Klasse hier beschrieben, auf diese wird im Verlauf des Kapitels noch eingegangen.

In dem Ordner mit dem Namen *OSGI-INF* sind die *Components* hinterlegt, diese enthalten die *Metadaten* über die zu importierenden/exportierenden *Services*. Beispielhaft wird der Inhalt des *IHeizungsregelungComponent*, welches die *Services* des *DeviceControllers* exportiert, gezeigt.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
3  name="org.wiedas.fhd.devices.heizungsregelung.IHeizComponent">
4  <implementation
5  class="org.wiedas.fhd.devices.heizungsregelung.DeviceController"/>
6  <service>
7  <provide
8  interface="org.wiedas.fhd.devices.heizungsregelung.intf.IHeizungsregelung"/>
9  </service>
10 </scr:component>
```

Listing 3: *IHeizungsregelung Component*

Der Name des *Components* in Zeile 3 ist frei wählbar, sollte allerdings einmalig sein. Andernfalls gibt es keine Fehlermeldung, *OSGi* stellt den *Service* stattdessen einfach nicht zur Verfügung.

Die Zeilen 4-5 enthalten den Namen der Klasse, die das Interface implementiert und die Methoden umsetzt. Von dieser Klasse wird dann vom Framework eine Instanz erzeugt, auf welche über das Interface, welches in den Zeilen 6-9 definiert ist, von anderen *Bundles* zugegriffen werden kann.

Auf den ersten Blick könnte man auf den Gedanken kommen, dass in einem *Component* mehrere *Services* von verschiedenen Klassen exportiert werden können. *Eclipse* bietet in der Formularansicht der *Components* auch die Möglichkeit, mehr als einen *Service* zur Verfügung zu stellen. Dies sollte man jedoch nicht tun, da auch dies zu keiner Fehlermeldung führt und die *Services* stattdessen nicht zur Verfügung stehen, da jedes *Component* immer nur eine Klasse, die die *Services* implementiert, beinhaltet.

In den nächsten Unterkapiteln werden die einzelnen Klassen des *Device-Bundles* vorgestellt.

4.3.1.1 Activator

Die *Activator*-Klasse implementiert das *OSGi*-Interface *BundleActivator*, welches Methoden beinhaltet, die aufgerufen werden, wenn das *Bundle* im Framework installiert oder deinstalliert wird. Das Klassendiagramm ist in Abbildung 16 zu sehen. Jedes *Bundle* kann eine *Activator*-Klasse beinhalten, in diesem Fall ist aber nur im *Device* eine enthalten.

[8] Daten die dem Framework mitteilen, welche Komponenten vorhanden sind und wie sie heißen.

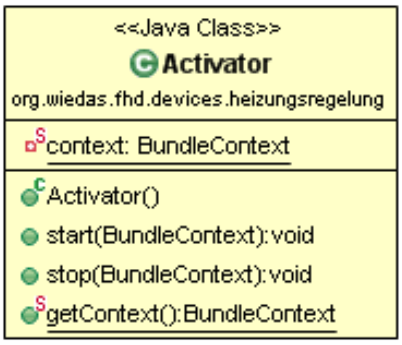


Abbildung 16: *Activator* Klasse

4.3.1.1.1 start()

Diese Methode wird vom Framework aufgerufen, wenn es das *Bundle* lädt. Hier können also Initialisierungen vorgenommen werden. In diesem Fall erfolgt lediglich eine Ausgabe auf der Konsole mit der Mitteilung „`LOGISCHES GERÄT :: Heizungsregelung Device gestartet`“.

4.3.1.1.2 stop()

Wenn das Framework das *Bundle* deinstalliert, wird diese Methode aufgerufen. Momentan wird lediglich die Mitteilung „`LOGISCHES GERÄT :: Heizungsregelung Device gestoppt`“ ausgegeben.

4.3.1.1.3 getContext()

Beim Starten des *Bundles* wird ihm vom Framework ein *BundleContext* übergeben. Der *BundleContext* dient in *OSGi* als Schnittstelle, über die das *Bundle* auf die Methoden des Frameworks zugreifen kann.

4.3.1.2 DeviceController, IHeizungsregelung

Der *DeviceController* implementiert die Methoden des Interfaces *IHeizungsregelung* und enthält die Umsetzung der von außen zugänglichen *Services* des *Bundles*. Außerdem beinhaltet er eine statische *ArrayList* vom Typ *ThermostatBean*, in der alle dem System bekannten Thermostate mit allen ihren Eigenschaften hinterlegt sind. Das *ThermostatBean* entspricht dabei in seinem Aufbau einem *JavaBean*[9], auf das *ThermostatBean* wird später noch eingegangen. Abbildung 17 zeigt das Klassendiagramm des *DeviceControllers*.

[9] Eine Klasse, die hauptsächlich der Speicherung von Daten dient. Charakteristisch sind *getter*- und *setter*-Methoden, mit denen auf die Attribute zugegriffen werden kann.

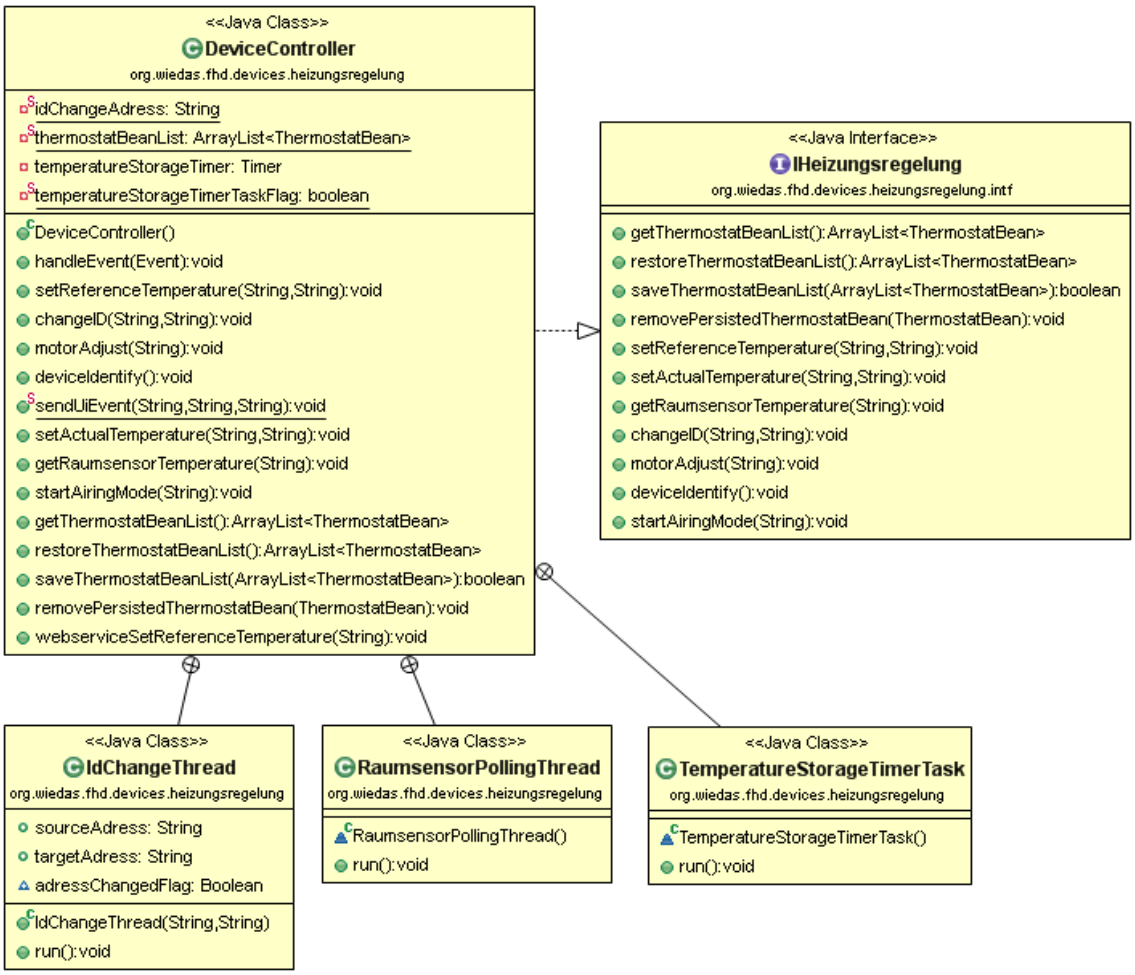


Abbildung 17: *DeviceController* Klasse

Folgend werden kurz die Methoden der Klasse und ihr Zusammenhang beschrieben.

4.3.1.2.1 handleEvent

Die Methode *handleEvent* wird aufgerufen, wenn vom *IP-Connector Bundle* ein *ipevent* verschickt wird. Die Zuordnung des Events zum *EventHandler* geschieht über den *EventAdmin Service* von *OSGi*, welcher, wie jeder andere *Service* auch, über das *Component* und den *ServiceBinder* eingebunden wird, dazu später mehr. Das *ipevent* enthält die ID des Gerätes (z.B. „*wm*“ für den Wassermelder oder „*ht*“ für das Heizkörperthermostat), die Adresse und die Daten die empfangen wurden. Fast alle Rückmeldungen der Thermostate werden in dieser Methode ausgewertet. Wird eine Nachricht von einem Thermostat empfangen, wird die Liste mit den bekannten Thermostaten durchlaufen. Stimmt die sendende Adresse mit einer bereits bekannten überein, wird die Nachricht ausgewertet und entsprechend reagiert:

- Beim Empfang des Strings „`ht_act_temp_XX`“ wird die Temperatur XX als aktuell gemessene Temperatur des Thermostats eingetragen. Diese Nachricht wird vom Thermostat verschickt, wenn es entweder seine aktuelle Temperatur gemessen hat oder wenn von außen eine aktuelle Temperatur zugewiesen wurde.

- Wenn bei einem Thermostat das Touch Panel gedrückt und die Solltemperatur geändert wurde, wird „`touch_temp_XX`“ verschickt, wobei XX die Temperatur ist. Diese wird bei Empfang als aktuelle Solltemperatur eingetragen.

- Wenn ein Raumsensor seine aktuelle Temperatur verschickt hat wird hier überprüft, ob der Raumsensor in einem der Thermostate hinterlegt ist und wenn ja wird diese Temperatur als aktuelle eingetragen und dem Thermostat per Funk zugeschickt.

- Sobald entweder eine Nachricht eines Thermostats oder eine Temperatur eines Raumsensors empfangen wurde, wird diese Nachricht inklusive Adresse und Geräte-ID mittels eines *uievent*s an die *View* geschickt. Dies geschieht mit der Methode *sendUievent*.

4.3.1.2.2 setReferenceTemperature

Die Methode *setReferenceTemperature* schickt die übergebene Solltemperatur an das Thermostat mit der ebenfalls übergebenen Adresse. Das Thermostat zeigt daraufhin für fünf Sekunden die empfangene Temperatur an.

*4.3.1.2.3 changeID & IdChange*Thread

Bei der Kommunikation mit den Geräten über *6LoWPAN* und per *UDP* verschickten Nachrichten kann es vorkommen, dass Pakete nicht beim Empfänger ankommen. Generell ist das Heizkörperthermostat immer noch einsatzfähig, wenn eine gewünschte Solltemperatur nicht empfangen wurde oder die aktuelle Temperatur nicht im *OSGi*-Framework ankommt. Wenn allerdings bei der Adressvergabe ein Paket verloren geht, ist das Heizkörperthermostat nicht mehr ansprechbar, da das System nicht wissen kann, welche Adresse aktuell vorhanden ist. Um dies zu verhindern, wird im Falle einer Adressänderung der *IdChange*Thread gestartet, der folgenden Programmablauf hat:

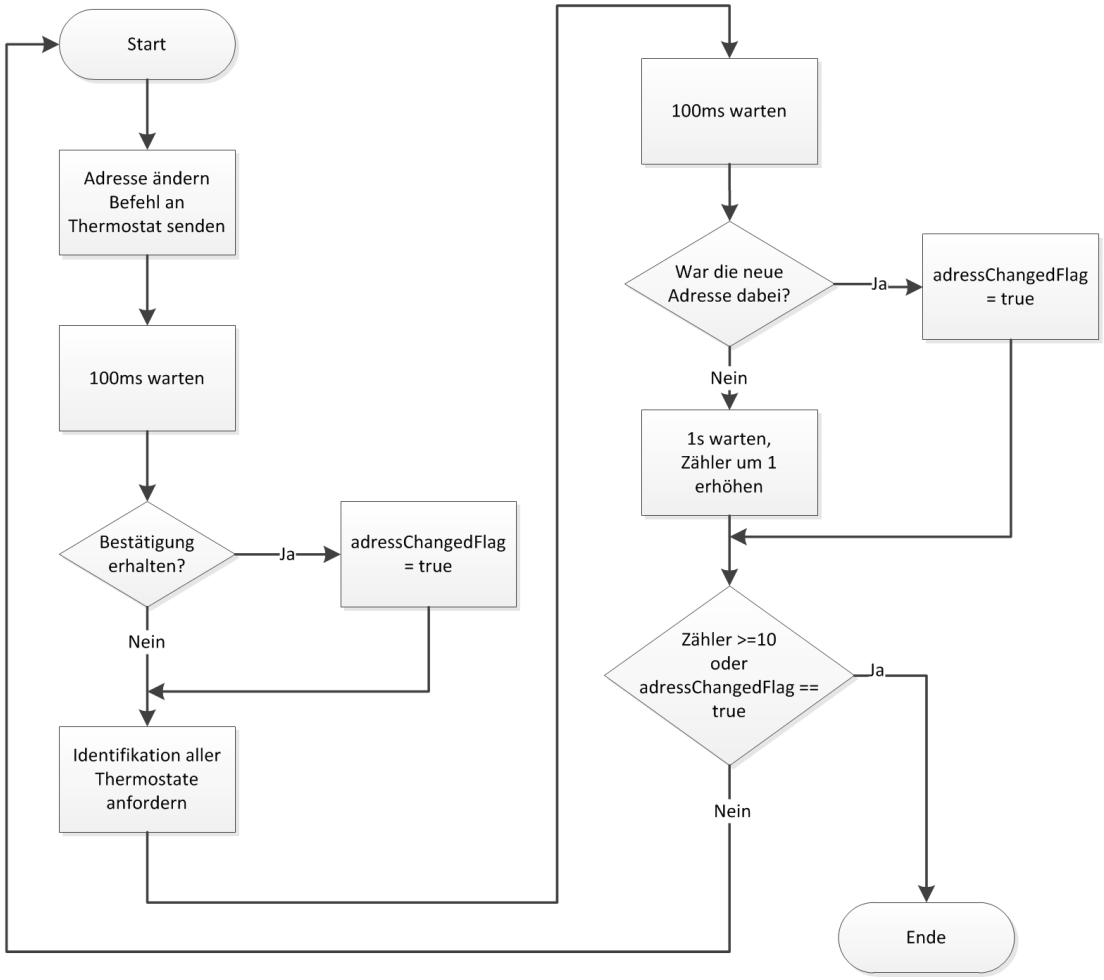


Abbildung 18:PAP *IdChangeThread*

Der Thread schickt als erstes einen Adressänderungsbefehl an das Thermostat. Das Thermostat bestätigt dies, wenn die Adresse geändert wurde, mit der Nachricht „ht:XXXX:ID Changed:YYYY“, wobei XXXX die neue und YYYY die alte Adresse ist. Der Thread wartet 100ms und überprüft, ob die Bestätigung empfangen wurde. Wenn ja, wird das Überprüfungsbit auf „*true*“ gesetzt. Anschließend wird eine Identifikation aller Thermostate angefordert und nach 100ms überprüft, ob die neue Adresse dabei war. Wenn ja, wird das Überprüfungsbit ebenfalls auf „*true*“ gesetzt. Anschließend wird eine Sekunde gewartet und das Ganze bis zu zehnmal wiederholt, außer das Überprüfungsbit ist gleich „*true*“. Damit ist ein Fehler durch ein verlorengegangenes Datenpaket sehr unwahrscheinlich.

4.3.1.2.4 motorAdjust

Die Funktion *motorAdjust* schickt an das Thermostat einen Justierungsbefehl. Das Thermostat stellt daraufhin die Regelung für die Dauer des Justierens ein und verfährt den Motor von Endanschlag zu Endanschlag um sich neue Referenzpunkte zur Positionsbestimmung des Motors zu holen.

4.3.1.2.5 deviceIdentify

Diese Methode verschickt den String „`ht:FFFF:ht_id_request`“, woraufhin alle Thermostate die diesen empfangen, mit dem String „`ht:XXXX:ht_id_ack`“ antworten, wobei XXXX die eigene Adresse ist. Darüber lassen sich unbekannte Thermostate im Netzwerk finden und es kann überprüft werden, ob bereits bekannte Thermostate noch eingeschaltet und/oder erreichbar sind.

4.3.1.2.6 sendUievent

Diese Methode versendet das *uievent* an die *Views*. Dieses kann von jeder *View* empfangen werden. Der *DeviceController* verschickt es immer dann, wenn entweder eine Nachricht des Heizkörperthermostats oder die Temperatur eines Raumsensors empfangen wurde. Dies geschieht im *Eventhandler* des *ipevent*s.

4.3.1.2.7 setActualTemperature

Diese Methode ermöglicht es, dem Thermostat mit der übergebenen Adresse eine „gemessene“ Temperatur aufzuzwingen. Empfängt das Thermostat diese, wird sie zwei Minuten lang als gemessene Temperatur für die Regelung verwendet und auf dem Display dargestellt. Wenn zwei Minuten lang keine externe Temperatur empfangen wurde, wird wieder der eigene Temperatursensor verwendet.

4.3.1.2.8 startAiringMode

Hiermit kann der Belüftungsmodus des Thermostats eingeschaltet werden. Das Thermostat stellt dann zehn Minuten lang die Regelung ein, schließt das Heizungsventil und zeigt ein Belüftungssymbol auf dem Display an.

4.3.1.2.9 getThermostatBeanList

Gibt die statische *ArrayList* vom Typ *ThermostatBean* zurück, in der alle dem System bekannten Thermostate hinterlegt sind. Sie wird bei Zugriffen auf die Datenbank immer aktualisiert, sodass immer die aktuellste Version vorhanden ist, mit der dann gearbeitet werden kann.

4.3.1.2.10 restoreThermostatBeanList, saveThermostatBeanList und removePersistedThermostatBean

Diese Methoden greifen auf die Methoden der *ThermostatPersistence* Klasse zu. Die Methode *restoreThermostatBeanList* gibt alle in der Datenbank persistierten Thermostate in einer *ArrayList* zurück, *saveThermostatBeanList* speichert alle in einer Liste übergebenen Thermostate in der Datenbank und *removePersistedThermostatBean* entfernt das übergebene Thermostat aus der Datenbank.. Die *ThermostatPersistence* Klasse wird im Kapitel 4.3.1.6 - ThermostatPersistence näher beschrieben. Die Änderungen werden in der *static* deklarierten lokalen *ArrayList thermostatBeanList* gespeichert. Dies

hat den Hintergedanken, dass wenn ein anderes *Bundle* auf die vorhandenen Thermostate zugreifen möchte, es immer mit der gleichen Liste arbeiten kann, welche im Hintergrund vom *Device* automatisch lokal und in der Datenbank synchronisiert wird. Der Zugriff erfolgt über die Methode *getThermostatBeanList*.

4.3.1.2.11 webServicesetReferenceTemperature

Stellt eine per *WebService* aufrufbare Methode zum Ändern der Solltemperatur eines Thermostats zur Verfügung. Die Beschreibung der Implementierung und Nutzung der *WebServices* befindet sich in der Bachelorthesis von Andreas Knoll (Knoll, 2013).

4.3.1.2.12 TemperatureStorageTimerTask

Um die erfassten Temperaturen der Thermostate zu speichern, wird jede Viertelstunde der Mittelwert gebildet, bevor sie persistiert werden. Dafür wurde dieser *TimerTask* geschrieben, welcher im Konstruktor des *DeviceControllers* gestartet wird und alle dreißig Sekunden überprüft, ob die Minuten der aktuellen Systemzeit glatt durch fünfzehn teilbar sind. Wenn ja, wird mit der Methode *addToTemperatureStorage* des *ThermostatBeans* der Mittelwert der bisher empfangenen Temperaturen für alle Thermostate gebildet und, mit der aktuellen Systemzeit versehen, in der *temperatureStorageList* der jeweiligen Thermostate gespeichert. Das Speichern nur zur ganzen Viertelstunde sorgt dafür, dass der Speicherbedarf der Liste nicht zu groß wird, denn erstens wird die Temperatur als String gespeichert, bei zwei Zeichen entspricht dies mit abschließender Null 3 Byte, und zweitens der *Timestamp* mit der aktuellen Systemzeit, der als primitiven Datentyp *long* (8 Byte) hat. Würde jede Minute einmal beides gespeichert, hätte man nach einem Tag bereits ~16kByte zusammen. Mit der Mittelwertbildung über eine Viertelstunde ist es dagegen nur ~ 1kByte. Außerdem ist durch die Mittelwertbildung der Temperaturverlauf glatter. Damit es, auch wenn mehrere *DeviceController* Klassen angelegt sein sollten, keine Konflikte zwischen den *TimerTasks* gibt, wird zur Verriegelung die *static*-Variable *temperatureStorageTimerTaskFlag* genutzt. Der *TimerTask* wird nur gestartet, wenn dieses „*false*“ ist und nach dem Start wird es auf „*true“* gesetzt. Durch die Verwendung des Schlüsselworts *static* greifen alle *DeviceController*-Instanzen auf die gleiche Variable zu, welche vom Compiler zuvor an einer festen Adresse im Arbeitsspeicher abgelegt wurde. Somit kann jede neue Instanz des *DeviceControllers* überprüfen, ob der *TimerTask* bereits vorhanden ist.

4.3.1.2.13 RaumsensorPollingThread

Dieser Thread wird ebenfalls im Konstruktor gestartet und durchläuft die Liste aller bekannten Thermostate und verschickt an jeden Raumsensor, der bei einem Thermostat hinterlegt ist, den String „`temp:XXXX:request`“, wobei XXXX für die Adresse des Raumsensors steht. Bei einer Übereinstimmung der Adresse mit der eigenen, antwortet der Raumsensor anschließend mit seiner aktuellen Temperatur. Diese wird im *Eventhandler* des *ipevent*s dem entsprechenden Thermostat in der Liste zugeordnet und dem Thermostat mit der Methode *setActualTemperature* zugeschickt.

4.3.1.3 Die Device-ServiceBinder

Die beiden *ServiceBinder ServiceBinderEventAdmin* und *ServiceBinderISocketCommunication* binden die *Services* des vom Framework zur Verfügung gestellten *EventAdmins* und des *IP-Connectors* für die *6LoWPAN* Kommunikation ein. Ihre Klassendiagramme sind in Abbildung 19 zu sehen.

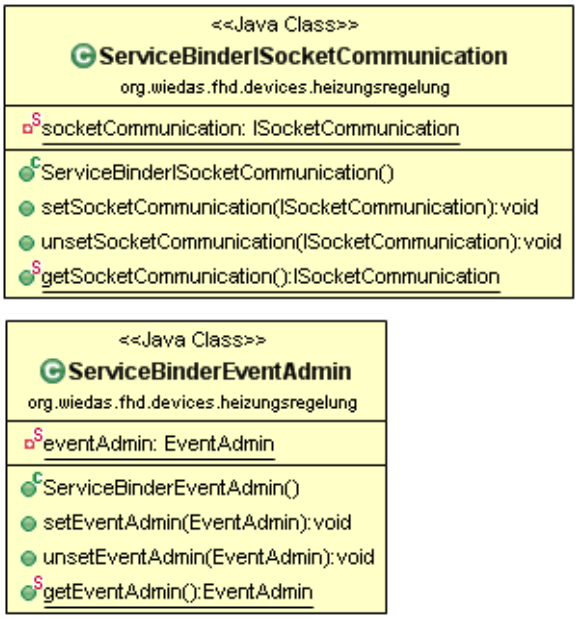


Abbildung 19: *ServiceBinder* Klassen im *Device*

Der Aufbau einer *ServiceBinder* Klasse wird im folgenden Listing 4 beispielhaft am *ServiceBinderISocketCommunication* gezeigt:

```
1  package org.wiedas.fhd.devices.heizungsregelung;
2  import org.wiedas.fhd.connector.ip.intf.*;

3  public class ServiceBinderISocketCommunication {
4    private static ISocketCommunication socketCommunication = null;

5    public void setSocketCommunication(ISocketCommunication
6    socketCommunication){
7        ServiceBinderISocketCommunication.socketCommunicationl =
8        socketCommunication;
9    }

10   public void unsetSocketCommunication(ISocketCommunication
11   socketCommunication){
12       ServiceBinderISocketCommunication.socketCommunication =
13       socketCommunication;
14   }

15   public static ISocketCommunication getSocketCommunication() {
16       return socketCommunication;
17   }
18 }
```

Listing 4: *ServiceBinderISocketCommunication*

Die *ServiceBinder* Klasse in *OSGi* ist im Prinzip ein *JavaBean*, das eine Instanz des Interfaces enthält, in dem die zu verwendenden *Services* deklariert sind. Beim Starten des *Bundles* werden die *ServiceComponents* des jeweiligen *Bundles* ausgeführt (dazu müssen diese im *Manifest* hinterlegt sein). In den *Components* steht wiederrum der *ServiceBinder* für den *Service*, welches Interface er benötigt und welches seine *getter*- und *setter*-Methoden sind. Die Interface-Instanz wird im *ServiceBinder* als *static* Variable gespeichert, damit mittels der Methode *getSocketCommunication()* auf die gleiche In-

terface-Instanz zugegriffen werden kann. Möchte man beispielsweise auf den *Service* zugreifen, mit dem man unverschlüsselt Daten per *UDP-Broadcast* versenden kann, kann man dann mit folgender Codezeile darauf zugreifen:

```
ServiceBinderISocketCommunication.getSocketCommunication().send_plain(…);
```

4.3.1.4 ThermostatBean

Das *ThermostatBean* stellt eine Struktur dar, in der alle Eigenschaften und Statuswerte eines Thermostates hinterlegt werden können. Von diesem wird im *DeviceController* eine statische *ArrayList* angelegt, um beliebig viele Thermostate verwalten zu können. In Abbildung 20 ist das dazugehörige Klassendiagramm zu sehen. Beim *ThermostatBean* wird nicht auf jede einzelne Methode eingegangen, da die meisten nur *getter-*/*setter*-Methoden für die Eigenschaften sind. Daher werden stattdessen die Attribute beschrieben.

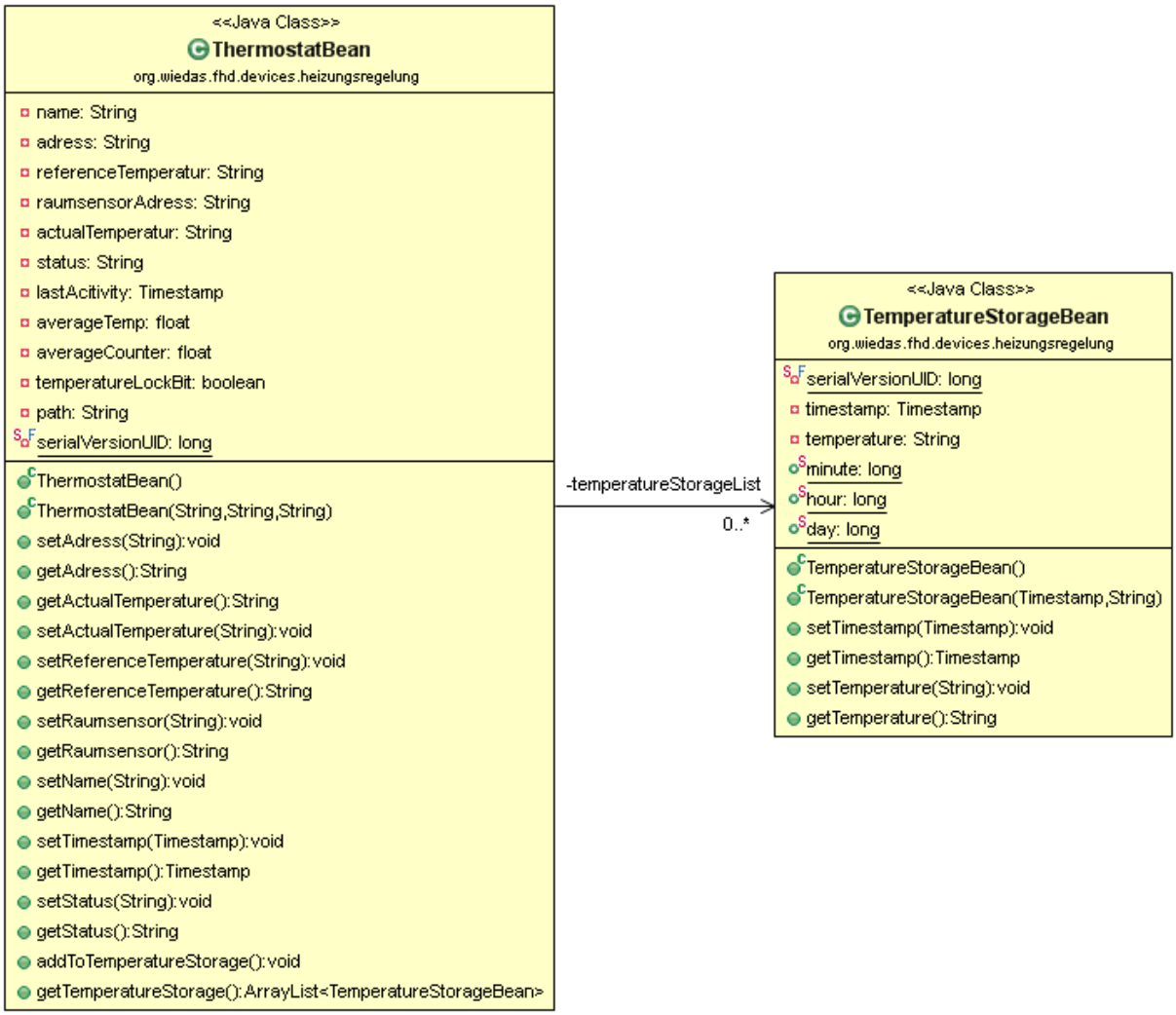


Abbildung 20: *ThermostatBean* Klasse

4.3.1.4.1 Default Konstruktor ThermostatBean()

Der Default Konstruktor muss in dieser Klasse vorhanden sein, damit sie mithilfe von *JPA* in der Datenbank persistiert werden kann. Die Datenbankanbindung wird im Kapitel 4.3.2 - Der Datenbankzugriff beschrieben. In ihm wird der aktuelle Status des Thermostats auf „*unknown*“ gesetzt und die aktuelle Temperatur gelöscht.

4.3.1.4.2 Überladener Konstruktor ThermostatBean (String, String, String)

Dieser Konstruktor dient dazu, beim Hinzufügen eines neuen Thermostats die Parameter Raumname, Adresse und Raumsensoradresse nicht extra mit den *Setter*-Methoden setzen zu müssen, sondern sie stattdessen direkt dem Konstruktor übergeben zu können.

4.3.1.4.3 adress

Die vierstellige Geräteadresse des Thermostats liegt in hexadezimaler Form vor, die Initialadresse, mit der sich „frisch" programmierte Thermostate anmelden, ist *FFFF*. Durch die automatische Adressvergabe wird die erste nicht vergebene Adresse gesucht und diese an das Thermostat vergeben. Dieses Attribut wird in der Datenbank persistiert.

4.3.1.4.4 actualTemperature

Die gemessene Temperatur des Thermostats. Wurde dem Thermostat ein Raumsensor zugeordnet und es wird von diesem eine Temperatur empfangen, wird stattdessen diese verwendet. Dieses Attribut wird **nicht** in der Datenbank persistiert.

4.3.1.4.5 referenceTemperature

Die aktuell eingestellte Solltemperatur des Thermostats. Dieses Attribut wird in der Datenbank persistiert.

4.3.1.4.6 raumsensorAdress

Die Adresse des Raumsensors, dessen gemessene Temperatur anstatt der des Thermostats verwendet werden soll. Sie ist wie die Adresse des Thermostats eine vierstellige Hexadezimalzahl. Dieses Attribut wird in der Datenbank persistiert.

4.3.1.4.7 name

Der Name des Raumes, in dem sich das Thermostat befindet. Es ist ein frei verwendbarer String ohne Einschränkungen. Dieses Attribut wird in der Datenbank persistiert.

4.3.1.4.8 lastActivity

Ein *Timestamp*, auf den über *setTimestamp* und *getTimestamp* zugegriffen werden kann. Wird immer auf die aktuelle Systemzeit gesetzt, wenn Daten von dem Thermostat empfangen werden. Wurden länger als fünf Minuten keine Daten empfangen, wird der Status des Thermostats automatisch auf „*unknown*" gesetzt. Dieses Attribut wird **nicht** in der Datenbank persistiert.

4.3.1.4.9 status

Der aktuelle Status des Thermostats. Er kann entweder „*active*", „*unknown*" oder „*deactivated*" sein. Er wird im Default Konstruktor auf „*unknown*" gesetzt. Sobald eine Nachricht des Thermostats empfangen wird, wird er auf „*active*" gesetzt und fünf Minuten nach der letzten Nachricht wieder auf „*unknown*". Der Status „*deactivated*" wird derzeit nicht genutzt. Dieses Attribut wird **nicht** in der Datenbank persistiert.

4.3.1.4.10 temperatureStorageList

ArrayList, die Instanzen der Klasse *TemperatureStorageBean* enthält. Mit der Methode *addToTemperatureStorage* können dem *ThermostatBean* gemessene Temperaturen mit einem Zeitstempel übergeben werden. Diese werden dann in der Liste gespeichert, welche mit *getTemperatureStorage* abgefragt werden kann. Die Liste dient der Darstellung des Temperaturverlaufs über der Zeit und wird in der Datenbank persistiert.

4.3.1.4.11 setActualTemperature

Diese Methode speichert den übergebenen String als aktuelle Temperatur ab. Außerdem werden alle übergebenen Werte bis zu fünfzehn Minuten lang aufsummiert.

4.3.1.4.12 addToTemperatureStorage

Diese Methode bildet den Mittelwert der seit dem letzten Aufruf erfassten Temperaturen des Thermostats und fügt diesen mit der aktuellen Systemzeit versehen der *temperatureStorageList* hinzu. Dabei wird das Datum der bereits gespeicherten Temperaturen geprüft und diese werden, wenn sie älter sind als dreißig Tage, entfernt. Dies soll einen zu großen Umfang der Datenbank verhindern, wenn viele Thermostate über einen langen Zeitraum betrieben werden sollten. Alle Daten, die entfernt werden, werden anschließend in einer Textdatei abgespeichert. Diese kann dann vom Nutzer je nach Bedarf genutzt oder gelöscht werden. Das Speichern in der Datei erfolgt mittels eines Java-*FileWriters* und ist im Listing 5 zu sehen:

```
1  private String path = "temperaturehistory.sav";
2  File file = new File(path);
3  FileWriter writer = new FileWriter(file, true);
4  for(TemperatureStorageBean removeStorage : removeList){
5      writer.write(name+" : "+removeStorage.getTimestamp().toString()+" : "
6      +removeStorage.getTemperature()+"°C"
7      +System.getProperty("line.separator"));
8  }
9  writer.flush();
10 writer.close();
```

Listing 5: Textspeicherung in einer Datei

Mit den in java.io. hinterlegten Klassen *File* und *FileWriter* kann in einer beliebigen Datei ein String gespeichert werden. Dazu muss ein *File* mit einem Pfad erstellt werden, in diesem Fall liegt die Datei, wie in Zeile 1 zu sehen ist, in keinem speziellen Verzeichnis, wodurch sie im *Eclipse* Ordner angelegt wird. Der Name der Datei lautet `temperaturehistory.sav`.

Diese muss nun dem *FileWriter* im Konstruktor übergeben werden, dies geschieht in Zeile 3. Das erste Übergabeparameter ist dabei die Datei, dass zweite stellt ein, ob die Datei neu erstellt werden soll, oder ob die zu schreibenden Daten an das Ende angehängt werden. Hier ist der Wert auf „*true*“ gesetzt, was die letztere Option aktiviert.

Anschließend wird in den Zeilen 4-8 eine Liste mit allen zu speichernden Daten durchlaufen und diese als String in dem *FileWriter* gespeichert.

Mit der *flush()* Methode des *FileWriters* werden sämtliche im Puffer enthaltene Daten sofort in den Ausgabestrom geschrieben. Dies geschieht um zu vermeiden, dass sich beim Schließen des *FileWriters* in Zeile 10 noch Daten in seinem Puffer befinden (Abts, 2013 S. 179).

4.3.1.5 TemperatureStorageBean

Die Klasse *TemperatureStorageBean* dient der Speicherung eines *Timestamps* und einer Temperatur als String und ist ebenfalls ein *JavaBean*. Das *ThermostatBean* legt, wie im vorigen Kapitel beschrieben, eine Liste dieser Klasse an um einen Temperaturverlauf aufzuzeichnen. Das Klassendiagramm sieht deshalb wie folgt aus:

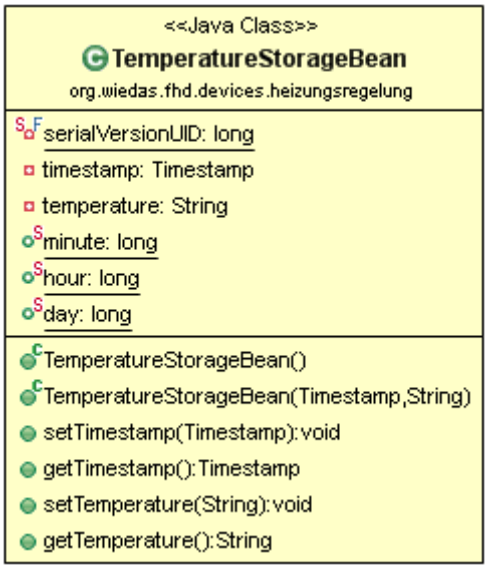


Abbildung 21: *TemperatureStorageBean* Klasse

Auf die Methoden wird nicht näher eingegangen, es sind lediglich *getter*/*setter* für die beiden Attribute: Das *Timestamp*-Attribut ist der Aufnahmezeitpunkt der jeweiligen Temperatur, die im *temperature*-Attribut gespeichert wird.

Weil die Klasse in einer Liste vorliegt, welche persistiert wird, muss sie das Markierungs-Interface *Serializable* implementieren. Dieses beinhaltet keine Methoden, braucht aber einen parameterlosen Konstruktor. Die Klassen innerhalb dieser Klasse müssen wiederrum serialisierbar sein, was bei *Timestamp* und String der Fall (Abts, 2013 S. 194) ist.

Ansonsten stellt diese Klasse noch die zu *Timestamp* kompatiblen Längen eines Tages, einer Stunde und einer Minute für Berechnungszwecke zur Verfügung.

4.3.1.6 ThermostatPersistence

Um den Zugriff auf die Datenbank zu vereinfachen, ist sämtliche Interaktion mit ihr in der *ThermostatPersistence* Klasse gekapselt. Das Klassendiagramm sieht folgendermaßen aus:

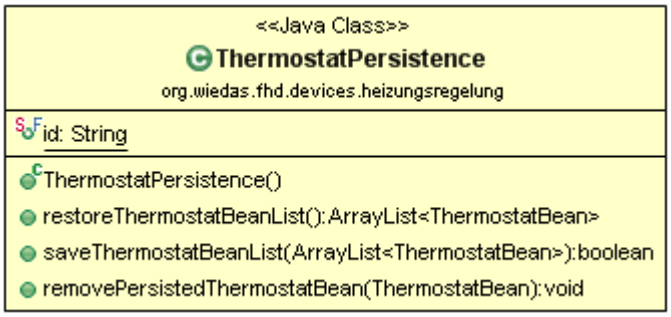


Abbildung 22: *ThermostatPersistence* Klasse

Die Datenbank ist eine *mySQL* Datenbank, auf welche mithilfe der Java Persistence *API* 2(*JPA*2) zugegriffen wird. Eine genauere Beschreibung der Datenbank und von *JPA* befindet sich im Kapitel 4.3.2 - Der Datenbankzugriff.

4.3.1.6.1 restoreThermostatBeanList

Die Methode *restoreThermostatBeanList* durchsucht die Datenbank nach vorhandenen Thermostaten und speichert diese in einer *ArrayList*, welche sie anschließend zurückgibt. Falls der Zugriff nicht möglich ist, wird eine leere Liste zurückgegeben.

4.3.1.6.2 saveThermostatBeanList

Hiermit werden sämtliche in der *ArrayList* übergebenen Thermostate in der Datenbank persistiert. Bei Erfolg wird „*true*“ zurückgegeben, andernfalls „*false*“.

4.3.1.6.3 removePersistedThermostatBean

Das übergebene Thermostat wird in der Datenbank gesucht und gelöscht.

Damit ist die Vorstellung des *Device Bundles* abgeschlossen und es folgt die Erklärung des Datenbankzugriffs.

4.3.2 Der Datenbankzugriff

Um die Daten der bekannten Thermostate der Anwendung auch nach dem Schließen verfügbar zu machen, müssen sie dauerhaft gespeichert und wiederhergestellt werden können.

Häufig werden zur Speicherung und Verwaltung großer Datenmengen Datenbanken eingesetzt. Diese haben gegenüber dem einfachen Schreiben in eine Textdatei einige Vorteile, wie z.B. Methoden für die Suche nach bestimmten Objekten, Sortieren von Objekten und eine Benutzerverwaltung. Da im *WieDAS*-Forschungsprojekt bereits für die Benutzerverwaltung der Anwendung eine *mySQL* Datenbank eingesetzt wurde, wurde entschieden diese auch zur Speicherung der Thermostate zu nutzen.

mySQL wird von der Oracle Corporation entwickelt und ist ein Open-Source Datenbankverwaltungssystem. Im Forschungsprojekt wird dies mit dem Freeware-Tool *XAMPP* umgesetzt, welches einen *Apache Webserver* mit einer darauf laufenden *mySQL* Datenbank beinhaltet. Da die Realisierung der Datenbank nicht Teil dieser Bachelorthesis war, wird aber nicht weiter darauf eingegangen

Die tatsächliche Umsetzung basiert aus praktischen Gründen auf dem bereits vorhandenen Zugriff auf die Datenbank bei der Benutzerverwaltung. Dieser wurde ursprünglich von Thomas Schmitz, einem ehemaligen Mitarbeiter des Informatiklabors, erstellt und basiert wiederrum auf einem Online-Tutorial (Vogel, 2012). Nachfolgend wird nur kurz auf den Datenbankzugriff eingegangen, da das Hauptaugenmerk auf der Anbindung der Heizungsregelung in das Funknetzwerk mittels *OSGi* lag.

Der Zugriff auf die *mySQL* Datenbank erfolgt mittels der *Java Persistence API* 2.0(*JPA* 2). Dabei handelt es sich um eine Programmierschnittstelle, die mithilfe von *Metadaten* (z.B. in einem speziellen *XML-File*) jedes beliebige Objekt, das nicht *final* oder *static* ist, persistieren kann. Dazu muss keine Änderung am Quellcode vorgenommen werden und die persistierten Daten verbrauchen den gleichen Speicherplatz, wie die Objekte, die sie darstellen (Keith, et al., 2009 S. 14).

Die Referenzimplementierung für *JPA* 2 ist *EclipseLink*, ein Open-Source Framework, welches außer *JPA* noch andere Standards für Datenbank- und Dateizugriffe unterstützt. Die beiden für *EclipseLink* erforderlichen Dateien sind in dem Projekt im Ordner *lib* untergebracht, dieser ist in Abbildung 23 noch einmal zu sehen. Die beiden Dateien sind die *EclipseLink.jar* und die *javax.persistence_1.0.0.jar*. Die dritte Datei in dem Ordner, *mySQL-connector-java-5.1.12-bin.jar* ist ein *JDBC* (**J**ava **D**ata**b**ase **C**onnectivity)-Treiber für *mySQL*.

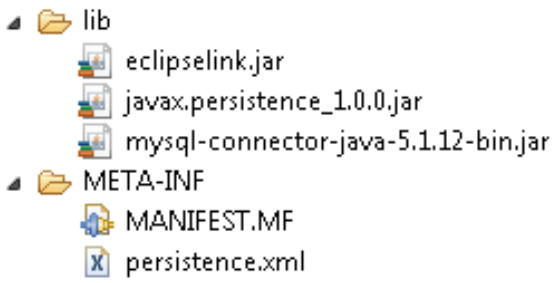


Abbildung 23: *EclipseLink* Dateien / *JDBC* Treiber

4.3.2.1 persistence.xml

Um *EclipseLink* mit der Datenbank verbinden zu können, sind einige *Metadaten* erforderlich. Diese sind in einer *XML*-Datei mit dem Namen *persistence.xml* untergebracht, welche ebenfalls in Abbildung 23 erkennbar ist.

Im folgenden Listing 6 ist der Inhalt der Datei dargestellt:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
4  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
5  <persistence-unit name="ThermostatBean">
6  <class>org.wiedas.fhd.devices.heizungsregelung.ThermostatBean</class>
7     <properties>
8        <property name="javax.persistence.jdbc.driver"
            value="com.mysql.jdbc.Driver" />
9        <property name="javax.persistence.jdbc.url"
            value="jdbc:mysql://localhost/aal_heizungsregelung" />
10       <property name="javax.persistence.jdbc.user" value="root" />
11       <property name="javax.persistence.jdbc.password" value="" />

12       <!-- EclipseLink should create the database schema automatically -->
13       <!-- property name="eclipselink.ddl-generation"
            value="create-tables" /-->
14       <property name="eclipselink.ddl-generation.output-mode"
            value="database" />
15    </properties>
16 </persistence-unit>
17 </persistence>
```

Listing 6: *persistence.xml*

Die für den Anwender wichtigen Zeilen sind die Zeilen 5-13. In Zeile 5 steht der Name der *Persistence-Unit*, welche dem Tabellennamen in der Datenbank entspricht und Zeile 6 spezifiziert die Klasse, die persistiert werden soll.

In den Zeilen 7-16 stehen nun die Eigenschaften der Datenbankverbindung. Die erste Eigenschaft in Zeile 8 spezifiziert den Treiber, mittels dessen auf die Datenbank zugegriffen wird. In diesem Fall der *mySQL* Treiber der *Java Database Connectivity*, der im Ordner *lib* hinterlegt wurde.

Die nächste Eigenschaft in Zeile 9 beschreibt die URL, unter der die Datenbank zu finden ist. Diese ist in diesem Fall lokal auf dem Rechner gespeichert, weshalb hier als Pfad „`//localhost/aal_heizungsregelung`“ steht. Der Name der Datenbank ist somit *aal_heizungsregelung*. Wenn

mehrere Applikationen im selben Netzwerk auf die gleiche Datenbank zugreifen sollten, könnte hier statt *localhost* die *IP* des Datenbankservers stehen.

Der Zugriff auf die Datenbank erfordert die Übergabe des Benutzernamens und des Passwortes, diese stehen in den Zeilen 10 und 11. Der Benutzername „root“ und ein leeres Passwort sind die Standardeinstellungen der Datenbank und sind sehr unsicher, weshalb sie bei einer tatsächlichen Anwendung in einem Haushalt geändert werden sollten. Ansonsten hat jeder, der einen Netzwerkzugang hat, freien Zugriff auf sämtliche persistierten Daten.

Mit dem auskommentierten Code in Zeile 13 kann man, wenn diese noch nicht besteht, die Tabelle innerhalb der Datenbank von *EclipseLink* erzeugen lassen. Wenn sie schon besteht führt diese Zeile zu einer *Exception*. Damit *JPA* die ausgewählte Klasse *ThermostatBean* nun persistieren kann, müssen in dieser noch Annotationen hinzugefügt werden.

4.3.2.2 Annotationen

Prinzipiell kann jedes Attribut einer Klasse, wenn es nicht *final* oder *static* ist, mit *JPA* in einer Datenbank gespeichert werden. Wenn dieses Element eine Liste einer selbst erstellten Klasse ist, wie im Fall der *TemperatureStorageBean* Liste im *ThermostatBean*, muss diese allerdings zusätzlich noch das Markierungs-Interface *Serializable* implementieren und alle ihre Attribute müssen ebenfalls serialisierbar sein. Die in der Bachelorthesis eingesetzten Annotationen werden im Folgenden anhand des *ThermostatBeans* gezeigt:

```
1   import javax.persistence.Entity;
2   import javax.persistence.Id;
3   import javax.persistence.Transient;

4   @Entity
5   public class ThermostatBean implements Serializable {
6   @Id private String name;
7   private String adress;
8   private String referenceTemperatur="20";
9   private String raumsensorAdress = "default";
10  private ArrayList<TemperatureStorageBean> temperatureStorageList
11  = new ArrayList<>();
12  @Transient private String actualTemperatur;
13  @Transient private String status;
14  @Transient private Timestamp lastAcitivity;
```

Listing 7: *JPA* Annotationen

Damit die Annotationen bekannt sind, muss die Klasse diese erst mittels der *javax.persistence.jar* Datei importieren. Dies geschieht in den Zeilen 1-3.

Die Klasse ist eine sogenannte *Entity* (ein zu speicherndes Objekt), kann also zu einer Zeile in einer Tabelle werden, die in einer Datenbank persistiert ist. Dies wird *JPA* mittels der in Zeile 4 zu sehenden „`@Entity`” Annotation mitgeteilt.

Um ein Objekt in der Datenbank zu speichern und später wieder finden zu können, muss es eine eindeutige ID besitzen. Diese kann entweder von *JPA* automatisch generiert werden oder vorgegeben werden. Im Falle des *ThermostatBeans* wird der Raumname, der bei jedem Thermostat einzigartig ist, als ID benutzt. Um eine Variable als ID zu kennzeichnen, muss die „`@Id`” Annotation davor stehen.

Mit diesen wenigen Einträgen können alle Attribute der Klasse in der Datenbank persistiert werden. Weil es aber nicht immer sinnvoll ist, jedes Attribut in der Datenbank zu sichern (z.B. den Aktivitätsstatus oder die aktuelle Temperatur), kann man diese mit „`@Transient`“ von der Speicherung ausschließen, wie in den Zeilen 12-14 zu sehen.

Der tatsächliche Zugriff auf die Datenbank kann jetzt mit dem *EntityManager* vorgenommen werden.

4.3.2.3 EntityManager

Die in der *persistence.xml* Datei hinterlegten Daten werden nun von *EclipseLink* genutzt, um eine *EntityManagerFactory* Klasse zur Verfügung zu stellen, welche den Zugriff auf die Datenbank im Code ermöglicht. Um einen Überblick zu bekommen, sind die Beziehungen der *JPA* Konzepte in Abbildung 24 dargestellt:

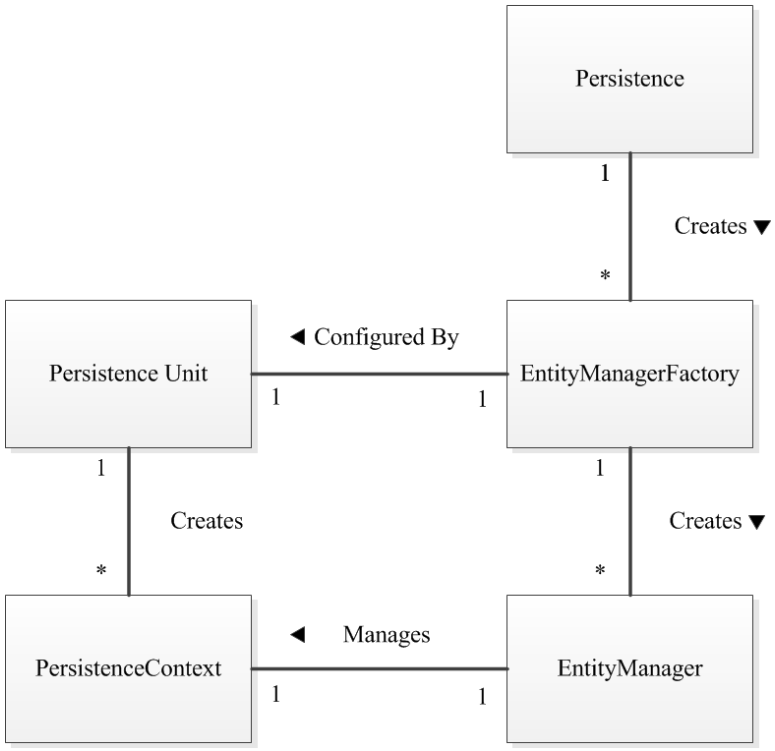


Abbildung 24: Beziehungen der *JPA* Konzepte (Keith, et al., 2009 S. 23)

Hier wird deutlich, dass für jede *Persistence-Unit*, also jede Tabelle in der Datenbank, eine einzige *EntityManagerFactory* zur Verfügung steht. Von dieser können allerdings beliebig viele *EntityManager* zum Zugriff auf die Datenbank generiert werden. Am besten lässt sich dies anhand eines Code-Beispiels (das keinen praktischen Nutzen hat) verdeutlichen:

```
1  EntityManagerFactory emf = Persistence.create-
   EntityManagerFactory("ThermostatBean");

2  EntityManager em = emf.createEntityManager();

3  ArrayList<ThermostatBean> thermostatList
   = new ArrayList<ThermostatBean>();

4  Query query = em.createQuery("SELECT t FROM ThermostatBean t");
5  thermostatList = new ArrayList<ThermostatBean>(query.getResultList());
```

```
6  for (ThermostatBean thermostat : thermostatBeanList) {
7      if(em.find(ThermostatBean.class, thermostat.getName())!=null){
8          em.getTransaction().begin();
9          em.remove(em.find(ThermostatBean.class, thermostat.getName()));
10         em.getTransaction().commit();
11     }
12     em.getTransaction().begin();
13     em.persist(thermostat);
14     em.getTransaction().commit();
15 }
16 em.close();
17 emf.close();
```

Listing 8: *EntityManager* Beispiel

Die *EntityManagerFactory* kann, wie in Zeile 1 zu sehen, durch die statische Methode `createEntityManagerFactory()` der Klasse *Persistence* generiert werden. Ihr wird als Parameter „`ThermostatBean`“ mitgegeben, welches der Name der *Persistence-Unit*, also der Tabelle in der Datenbank ist. Daraufhin kann in Zeile 2 der *EntityManager* von ihr erstellt werden.

Die *ArrayList* in Zeile 3 dient später der Speicherung der in der Datenbank persistierten Thermostate.

In diesem Beispiel wird davon ausgegangen, dass die Datenbank bereits besteht und Thermostate beinhaltet. Um z.B. beim Start der Applikation alle Thermostate aus der Datenbank auszulesen, kann ein *Query* genutzt werden. Dieser wird in Zeile 4 erstellt und gibt in Zeile 5 eine Liste zurück, in der die den Suchspezifikationen entsprechenden Objekte gespeichert sind. Die Sprache, in der dem *Query* die Suchspezifikationen mitgegeben werden ist die **J**ava **P**ersistence **Q**uery **L**anguage (*JPQL*). In diesem Falle werden alle vorhandenen Thermostate zurückgegeben.

Anschließend wird die Liste in den Zeilen 6-15 durchlaufen, und jedes Thermostat, das auch in der Datenbank ist (in diesem Fall also alle) wird anschließend gelöscht. Sowohl die *find()*, als auch die *remove()* Methode benötigen den Typ und die ID des jeweiligen Objekts um es zu finden oder zu löschen. Das Objekt ist in diesem Fall die *ThermostatBean* Klasse und die jeweilige ID der Raumname.

Danach wird jedes Objekt in der Liste mit der *persist()* Methode in Zeile 14 wieder in der Datenbank hinterlegt.

Zugriffe auf die Datenbank (bis auf einige Ausnahmen wie z.B. die *find()* Methode oder ein *Query*) müssen immer in einer *Transaction*, einem eigenen Arbeitsvorgang abgewickelt werden. Es muss außerdem darauf geachtet werden, dass immer nur ein Zugriff auf das gleiche Objekt in der Datenbank erfolgt. Die *Transaction* wird vom *EntityManager* z.B. in Zeile 9 gestartet und in Zeile 11 jeweils durchgeführt.

Zum Abschluss werden in Zeile 17-18 der *EntityManager* und die *EntityManagerFactory* geschlossen.

Damit kann nun erfolgreich auf die Datenbank zugegriffen werden, Daten gefunden, entfernt und persistiert werden. Dies reichte im Rahmen der Bachelorthesis völlig aus, weshalb das Thema hier nicht weiter vertieft wird.

Im nächsten Kapitel 4.3.3 - Die View wird auf das *View Bundle* und die Erstellung von Benutzeroberflächen mit *SWT* (dem **S**tandard **W**idget **T**oolkit)eingegangen.

4.3.3 Die View

Damit der Benutzer die Thermostate auch komfortabel bedienen kann, wurde eine grafische Benutzeroberfläche mit dem *Eclipse* Toolkit *SWT* entworfen. Das **S**tandard **W**idget **T**oolkit (*SWT*) ist eine Programmbibliothek zur *GUI*-Erstellung für *Eclipse*, die den Zugriff auf die nativen Benutzeroberflächen des Betriebssystems, auf dem die Anwendung läuft, ermöglicht. Es bietet viele Komfortfunktionen zur Erstellung der Oberflächen, so können die in der Designansicht auf der linken Bildschirmseite bereitstehenden Bedienelemente einfach per „*Drag & Drop*“ hinzugefügt werden. Soll nun eine Methode ausgeführt werden, wenn der Benutzer später mit den Elementen interagiert, so kann mit einem Rechtsklick auf das Element unter der Auswahl „*Add event handler*“ die Aktion ausgewählt werden, auf die reagiert werden soll. *SWT* fügt anschließend im Quellcode automatisch den entsprechenden *EventHandler* ein und diesem muss nur noch der auszuführende Code mitgegeben werden. Bei einem Mausklick auf das Benutzersteuerelement z.B. wird meistens das *widgetSelected* Event versandt.

Um die im Informatiklabor eingesetzte Bundlestruktur (vgl. Abbildung 14: *OSGi-Bundlestruktur* des Informatik Labors) einzuhalten wurde hierfür ein *View Bundle* erstellt. Die Projektstruktur ist wie folgt aufgebaut:

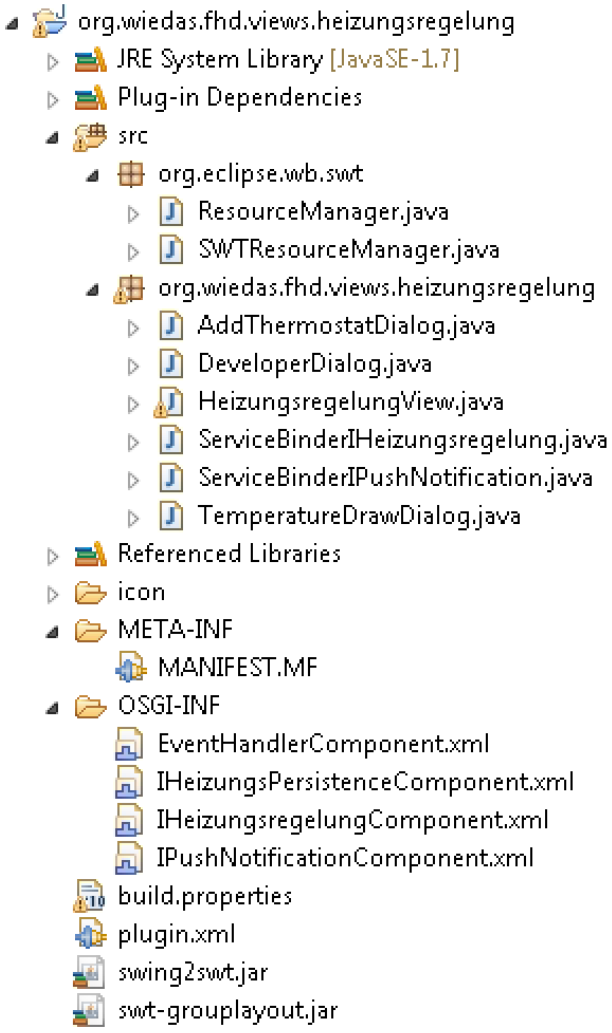


Abbildung 25: *View* Projektstruktur

Wie man sehen kann, ist die Struktur der des *Devices* recht ähnlich. Auch hier gibt es ein *Package* (das mit dem Namen *org.WieDAS.fhd.Views.heizungsregelung*) das die Umsetzung der Klassen und Methoden, sowie die *ServiceBinder* beinhaltet. Außerdem sind die bereits in Kapitel 4.3.1 beschriebenen *Components* und die *Manifest* Datei ebenfalls vorhanden.

Das *Package org.Eclipse.wb.SWT* enthält zwei *ResourceManager*, mit welchen Betriebssystemressourcen, wie z.B. Farben und Bilder, die in *SWT* Objekte eingebunden werden, verwaltet werden. Dadurch muss sich der Benutzer nicht mehr selbst um die korrekte Zuordnung und Entsorgung der Ressourcen zu kümmern. Der Ordner *icon* enthält die Bilder, die in der Anwendung zum Einsatz kommen und auf die mit dem *ResourceManager* zugegriffen werden kann.

Der tatsächliche Aufruf des *Bundles* geschieht nicht in der *View* oder dem *Device*. Es wird extern, von dem von Thomas Schmitz erstellten *rcpbasis Bundle*, aufgerufen. Dieses beinhaltet einen Container in den alle vorhandenen Benutzeroberflächen der Geräte hineingeladen werden und dort anschließend frei bewegt werden können. Dafür benötigt man allerdings noch die Bibliotheken *swing2SWT.jar* und *SWT-grouplayout.jar*. Die tatsächliche Anbindung geschieht durch die Datei *plugin.xml*, wobei die folgenden Zeilen die Spezifikation der zu ladenden *GUI* beinhalten:

```
1  <view
2      class="org.wiedas.fhd.views.heizungsregelung.HeizungsregelungView"
3      id="org.wiedas.fhd.views.heizungsregelung.Heizungsregelung"
4      name="Heizungsregelung">
5  </view>
```

Listing 9: *plugin.xml*

Als nächstes wird auf die Klassen des *.heizungsregelung Package*s eingegangen.

4.3.3.1 HeizungsregelungView

Die *HeizungsregelungView* ist die Hauptklasse des *View Bundles* und beinhaltet den größten Teil der Logik. Sie wird über die *plugin.xml* Datei in den Container für die Benutzeroberflächen der Geräte geladen und ist somit nach dem Starten der Anwendung sichtbar. Das Klassendiagramm sieht wie folgt aus:

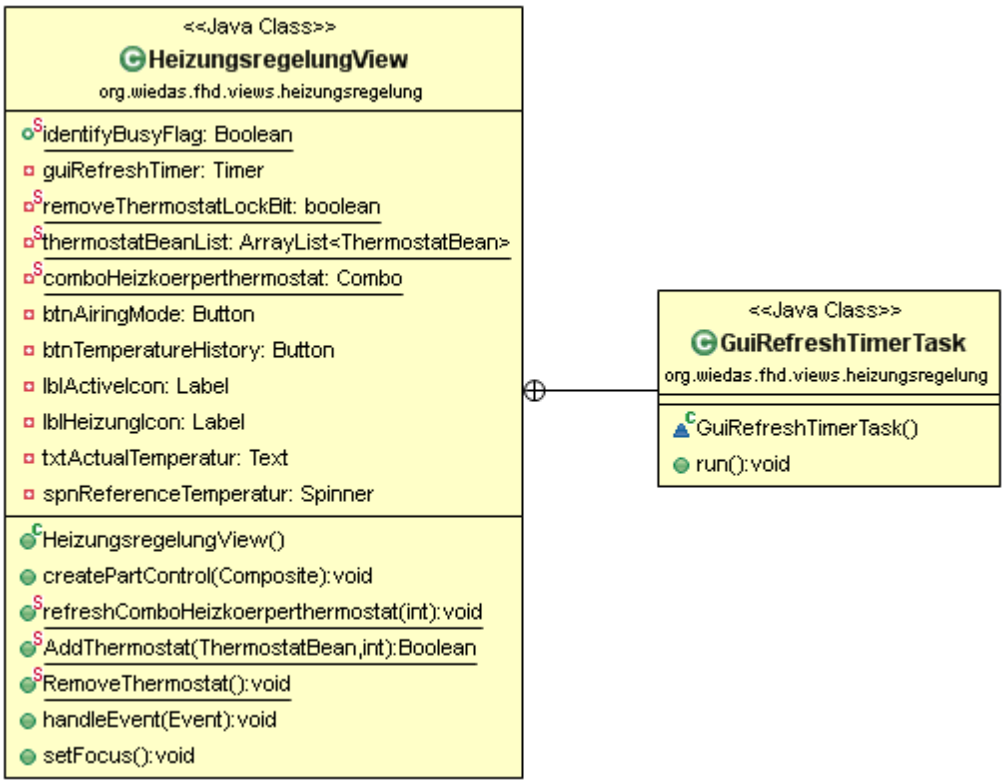


Abbildung 26: *HeizungsregelungView* Klasse

Im Folgenden wird zuerst auf die Attribute und Methoden eingegangen, anschließend folgt die Vorstellung der dadurch realisierten Oberfläche.

4.3.3.1.1 identifyBusyFlag

Das *identifyBusyFlag* wird während der Identifikation der Thermostate genutzt und verriegelt nach der Identifikation eines unbekannten Thermostats und dem darauffolgenden Öffnen des *AddThermostat-Dialogs* das Öffnen eines weiteren Dialogs bis dieser geschlossen wurde.

4.3.3.1.2 guiRefereshTimer

Dieser Timer startet zyklisch den *GuiRefreshTimerTask*, in dem alle Elemente, die Statusinformationen des aktuell ausgewählten Thermostats anzeigen, aktualisiert werden.

4.3.3.1.3 removeThermostatLockBit

Diese Variable ist, wie der Name schon sagt, ein *LockBit*, welches verhindert, dass während der Aktualisierung der *GUI* im *GuiRefreshTimerTask* ein Thermostat gelöscht werden kann. Dies führte ansonsten zu einer *NullPointer Exception*. Es wird daher innerhalb der Löschroutine gewartet, bis das *LockBit* durch die Aktualisierung der *GUI* wieder freigegeben wurde.

4.3.3.1.4 thermostatBeanList

Diese *ArrayList* vom Typ *ThermostatBean* beinhaltet die lokale Kopie der in der Datenbank persistierten Thermostate, um nicht bei jeder Aktion die Datenbank abfragen zu müssen. Sie wird bei Änderungen aber immer mit ihr synchronisiert. Das *ThermostatBean* wurde bereits in Kapitel 4.3.1.4 - ThermostatBean beschrieben.

4.3.3.1.5 Benutzersteuerelemente

Alle weiteren Attribute sind Benutzersteuerelemente, deren Status manipuliert werden muss, wie z.B. die Sichtbarkeit, die Aktivierung oder der Text. Weil sie normalerweise in der *createPartControl()* Methode deklariert werden, sind sie außerhalb dieser Methode dann nicht zugänglich. Daher wurden diese Benutzersteuerelemente zu Attributen umfunktioniert, um auch z.B. im *GuiRefreshTimerTask* auf sie zugreifen zu können.

4.3.3.1.6 HeizungsregelungView()

Im Konstruktor der Klasse wird über den *ServiceBinderIHeizungsregelung* auf den *Service getThermostatList* des *DeviceControllers* zugegriffen und somit auf die, im *DeviceController* gespeicherte, aus der Datenbank wiederhergestellte Liste der Thermostate.

4.3.3.1.7 createPartControl()

In dieser Methode werden der *View* alle Benutzersteuerelemente hinzugefügt. *SWT* fügt die automatisch generierten *EventHandler* für diese ebenfalls hier ein. Die *EventHandler* werden hier nicht explizit beschrieben, ihre Auswirkungen werden indirekt durch die Funktionsbeschreibung der Benutzeroberfläche geschildert.

4.3.3.1.8 refreshComboHeizkoerperthermostat()

Mit dem Aufruf dieser Methode kann man die *ComboBox*[10], über die der Nutzer die verschiedenen Thermostate (sofern vorhanden) auswählen kann, aktualisieren. Dazu werden die Raumnamen aller Thermostate in einem Array vom Typ String gespeichert und diese der *ComboBox* als Anzeigeelemente übergeben. Der Übergabeparameter der Funktion stellt das nach der Aktualisierung sichtbare Element ein. Ist er negativ oder größer als die Anzahl der Listenelemente, wird das letzte Listenelement ausgewählt.

4.3.3.1.9 AddThermostat()

Diese Methode ist *public* und *static* und somit auch ohne eine lokale Instanz der Klasse *HeizungsregelungView* ausführbar. Die Übergabeparameter sind eine Instanz eines *ThermostatBeans* und ein Index des Datentyps int. Soll ein bestehendes Thermostat überschrieben werden, so muss dessen Position in der Liste der Thermostate als Index übergeben werden, soll nichts überschrieben werden, muss der Index negativ sein. Am Anfang der Methode wird die Liste der bekannten Thermostate durchlaufen und überprüft, ob bereits ein Thermostat mit dem gleichen Namen oder der gleichen Adresse existiert. Wenn dies der Fall ist, wird eine *MessageBox* geöffnet, mit dem Hinweis das ein Namens-/Adresskonflikt erkannt wurde und ob das alte Thermostate ersetzt werden soll. Wenn dies bestätigt wurde, wird das Thermostat ersetzt und die *ComboBox* aktualisiert. Bei erfolgreicher Durchführung wird „*true*“ zurückgegeben, ansonsten „*false*“.

4.3.3.1.10 removeThermostat

Bei Aufruf dieser Methode wird eine *MessageBox* geöffnet, die den Benutzer fragt, ob er sich sicher sei, dass das Thermostat gelöscht werden soll. Wenn dies bestätigt wird, wird das aktuell in der *ComboBox* ausgewählte Thermostat sowohl aus der lokalen Liste, als auch aus der Datenbank entfernt. Dazu wird der *Service removePersistedThermostatBean*() des *DeviceControllers* benutzt.

4.3.3.1.11 handleEvent()

Diese Methode wird über den *EventHandlerComponent* beim Empfang des *uievent* aufgerufen.[11] Das *uievent* wird, nach dem Empfang von Daten von einem der Thermostate, vom *DeviceController* verschickt. Da sich dadurch eventuell Daten bezüglich eines Thermostats geändert haben, wird im *EventHandler* der *View* zuerst die aktuelle Liste der Thermostate des *Devices* abgerufen und gespeichert. Der folgende Programmablaufplan zeigt den Ablauf des *EventHandler*s:

[10] Eine Art Drop-Down-Menü, in diesem Fall nicht mehr durch den Benutzer editierbar.

[11] Dies funktioniert nach dem gleichen Prinzip wie bei dem *DeviceController* mit dem *ipevent*.

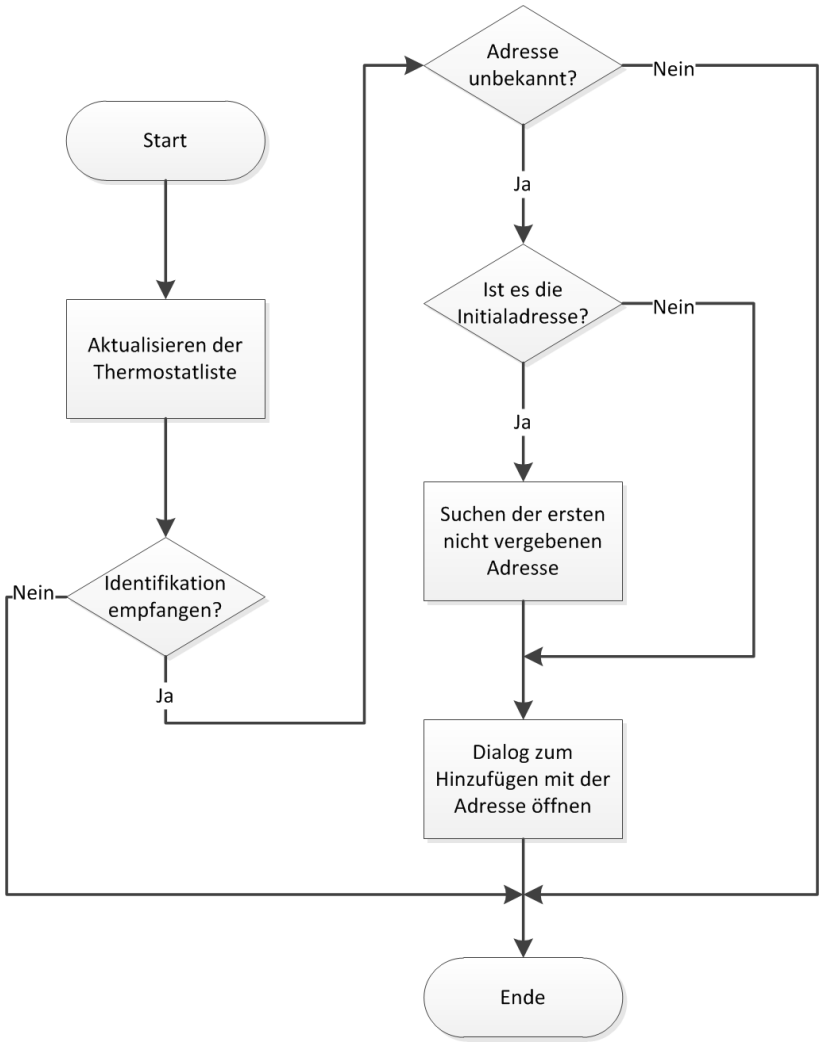


Abbildung 27: PAP *EventHandler uievent*

4.3.3.2 HeizungsregelungView - Benutzeroberfläche

Die in Abbildung 28 zu sehende Benutzeroberfläche ist die *View*, die beim Start des *View Bundles* vom *rcpbasis Bundle* aufgerufen wird, und somit diejenige, die der Benutzer beim Starten der Anwendung zu sehen bekommt.

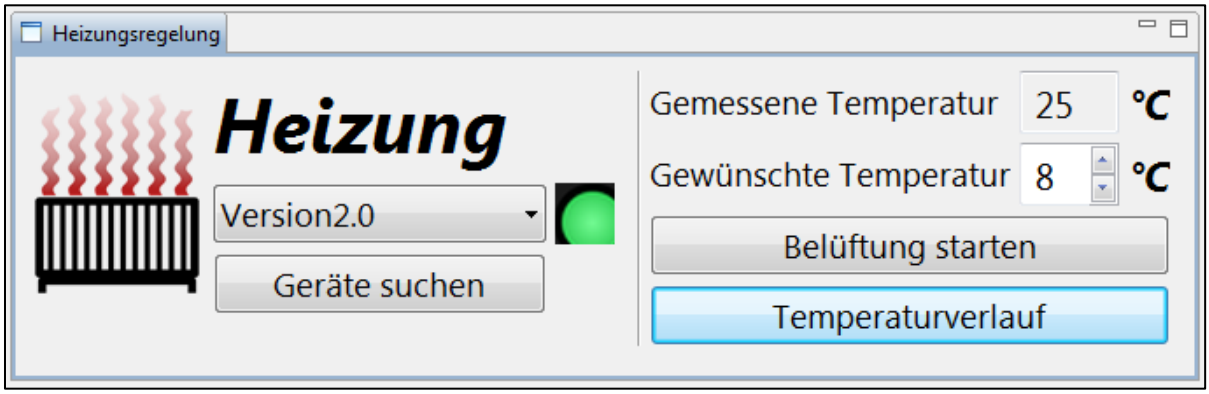


Abbildung 28: *HeizungsregelungView GUI*

Die Benutzersteuerelemente haben folgende Funktionen:

4.3.3.2.1 ComboBox

Die *ComboBox* ist die *Drop-Down* Auswahlschaltfläche für die Thermostate. In diesem Fall ist das Thermostat mit dem Raumnamen „Version2.0“ ausgewählt, es kann aber ein beliebiger String eingegeben werden, wie z.B. „Wohnzimmer“ oder „Bad“. Die Schaltfläche ist nur aktiviert, wenn auch Thermostate vorhanden sind. Alle Aktionen (bis auf einige in den Entwickleroptionen), wie z.B. die Erhöhung der Solltemperatur, werden nur an das hier ausgewählte Thermostat übertragen.

4.3.3.2.2 „Geräte suchen“ Knopf

Der Button mit der Aufschrift „Geräte suchen“ dient dem Hinzufügen neuer Thermostate in die Liste. Beim Drücken wird der *deviceIdentify() Service* des *DeviceControllers* aufgerufen, woraufhin sich alle Thermostate zurückmelden. Anschließend wird, wie in Kapitel 4.3.3.1.11 - *handleEvent*() beschrieben, automatisch eine Adresse vergeben. Mit dieser wird dann der *AddThermostatDialog* geöffnet.

4.3.3.2.3 Statusanzeige

Die Statusanzeige zeigt, wie der Name schon sagt, den aktuellen Status des ausgewählten Thermostats an. Die verschiedenen Status können sein:

- *active* – wird mit einem grünen Symbol angezeigt
- *unknown* – wird mit einem gelben Symbol angezeigt
- *deactivated* – wird mit einem roten Symbol angezeigt

Der *deactivated*-Status kann zwar angezeigt werden, wird derzeit aber nicht verwendet.

Wichtig: Ein Rechtsklick auf die Statusanzeige öffnet die Entwickleroptionen.

4.3.3.2.4 Gemessene Temperatur

Diese Anzeige ist zwar selbsterklärend, allerdings sollte noch erwähnt werden, dass die gemessene Temperatur des Thermostats nur angezeigt wird, wenn auch eine empfangen wurde. Ansonsten bleibt das Textfeld leer.

4.3.3.2.5 Gewünschte Temperatur

Hier kann der Nutzer seine Wunschtemperatur vorgeben, welche dann über den *Service setActualTemperature()* an das ausgewählte Thermostat geschickt wird. Wenn kein Thermostat in der Liste ist, ist die Schaltfläche deaktiviert. Die Eingabe kann nur zwischen 8°C und 25°C liegen, damit einerseits der Frostschutz gewährleistet wird und andererseits der Nutzer nicht versehentlich seine Wohnung zu stark aufheizt.

4.3.3.2.6 „Belüftung starten“ Knopf

Dieser Knopf ruft den *startAiringMode() Service* auf, wodurch das ausgewählte Thermostat zehn Minuten lang die Regelung deaktiviert und das Ventil zudreht. Wenn kein Thermostat in der Liste ist, ist die Schaltfläche deaktiviert.

4.3.3.2.7 „Temperaturverlauf“ Knopf

Durch einen Druck auf diesen Knopf wird eine Instanz des *TemperatureDrawDialog* geöffnet, welcher als Übergabeparameter das aktuell gewählte Thermostat bekommt und daraufhin dessen Temperaturverlauf darstellt. Wenn kein Thermostat in der Liste ist, oder weniger als zwei aufgezeichnete Temperaturen in der Temperaturliste des Thermostats sind, ist die Schaltfläche deaktiviert.

4.3.3.3 AddThermostatDialog

Die Klasse *AddThermostatDialog* erweitert die *SWT* Klasse *Dialog* und dient dem Hinzufügen von neuen Thermostaten zur Liste der bekannten. Dem entsprechend ist das Klassendiagramm auch recht übersichtlich:

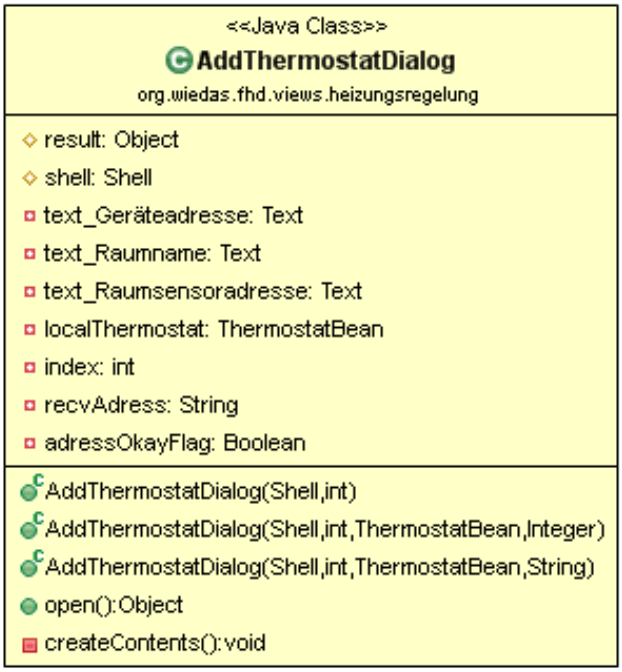


Abbildung 29: *AddThermostatDialog* Klasse

Im Folgenden wird zuerst auf die Attribute und Methoden eingegangen, anschließend folgt die Vorstellung der dadurch realisierten Oberfläche.

4.3.3.3.1 result

Das *result* ist der Rückgabewert der Methode *open()* der Klasse *Dialog*, welche hier erweitert wird. Er wird in diesem Fall aber nicht weiter verarbeitet und deshalb nicht genauer beschrieben.

4.3.3.3.2 Shell

Ein Objekt vom Typ *Shell* repräsentiert in *SWT* ein Fenster, wie auch der *Dialog* eines ist. In diesem wird das Layout des *AddThermostatDialogs* angelegt und dieses wird letzten Endes geöffnet, geschlossen, auf Aktivität überprüft und usw. Es ist also die Darstellungsfläche.

4.3.3.3.3 Textfelder

Die drei Textfelder sind die Eingabemaske für den Raumnamen, die Raumsensoradresse und die Geräteadresse.

4.3.3.3.4 localThermostat

Dies ist die lokale Kopie des übergebenen Thermostats. Sie wird zur Adressänderung genutzt und mit den neu eingetragenen Werten zurückgegeben.

4.3.3.3.5 index

Der Index kann vom aufrufenden Objekt im Konstruktor übergeben werden. Dies geschieht nur, wenn ein Thermostat geändert werden soll: Dann ist *index* die Position des Thermostats in der Liste und

nach dem Eintragen der gewünschten Werte in die Textfelder wird dieser Index genutzt um die Methode *AddThermostat()* der *HeizungsregelungView* aufzurufen, die das Thermostat an dieser Position dann überschreibt.

4.3.3.3.6 recvAdress

Dieser String vom aufrufenden Objekt übergeben werden und enthält die Adresse unter der das Thermostat erreichbar ist. Dies wird bei der automatischen Adressänderung genutzt, in dem die neue Adresse bereits im Thermostat eingetragen ist und die *recvAdress* die aktuelle enthält.

4.3.3.3.7 adressOkayFlag

In dieser Variablen vom Typ *boolean* wird der Rückgabewert der *AddThermostat()* Methode der *HeizungsregelungView* gespeichert. Diese wird aufgerufen, wenn der „Speichern“ Button gedrückt wurde. Der *Dialog* wird nur geschlossen, wenn der Rückgabewert „*true*“ ist. Der Rückgabewert „*false*“ kann z.B. auftreten, wenn schon ein Thermostat den gleichen Raumnamen hat und bei der folgenden *MessageBox* mit der Frage, ob man dieses überschreiben will auf „Abbrechen“ gedrückt wurde. In diesem Fall bleibt der *AddThermostatDialog* geöffnet.

4.3.3.3.8 Konstruktoren

Der Default Konstruktor wird in der Anwendung derzeit nicht eingesetzt. Der überladene Konstruktor *AddThermostatDialog(Shell, int, ThermostatBean, Integer)* wird genutzt um ein bestehendes Thermostat zu ändern, das Thermostat an der Stelle in der Liste, auf die der Integerwert zeigt wird dabei ersetzt. Der Konstruktor *AddThermostatDialog(Shell, int, ThermostatBean, String)* enthält die als String gespeicherte Adresse, unter der das Thermostat erreichbar ist. Dieser wird eingesetzt, um anschließend einen Adressänderungsbefehl an das entsprechende Thermostat zu senden.

4.3.3.3.9 open()

Dies ist eine erweiterte Methode der *SWT Dialog* Klasse, die das das tatsächliche Fenster, in dem das Layout aufgebaut wird, das *Shell*, öffnet und nach dem Schließen desselben durch Drücken entweder auf „Speichern“ oder „Abbrechen“ ein Objekt mit dem Status zurückgibt (*result*). Daher ist dies die Methode mit der man den *Dialog* tatsächlich öffnen kann.

4.3.3.3.10 createContents()

Innerhalb dieser Methode wird in allen *SWT Dialogs* oder *Views* das Layout angelegt, d.h. die Buttons, Textfelder usw. platziert, skaliert und mit *EventHandlern* versehen.

4.3.3.4 AddThermostatDialog – Benutzeroberfläche

Der *AddThermostatDialog* wird entweder aufgerufen, wenn von der *HeizungsregelungView* ein neues Thermostat erkannt wurde oder wenn in den Entwickleroptionen des *DeveloperDialogs* auf den „Ändern“ Knopf gedrückt wurde (mehr dazu im Kapitel 4.3.3.6 - DeveloperDialog – Benutzeroberfläche).

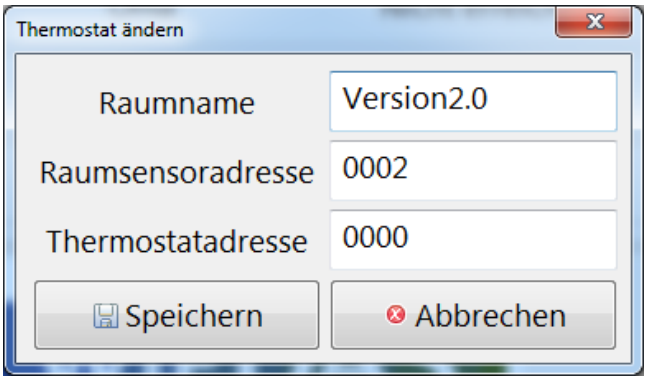


Abbildung 30: *AddThermostatDialog GUI*

Die drei Eingabefelder ermöglichen es, dem Thermostat einen Raumnamen, eine Raumsensoradresse und eine Thermostatadresse zuzuweisen.

4.3.3.4.1 Raumname

Der Raumname ist ein frei wählbarer String, in diesem Fall wurde „Version2.0“ als Name eingetragen. Es kann aber genauso gut „Wohnzimmer“ oder „Bad“ sein. Wird nichts eingegeben und auf „Speichern“ gedrückt, wird eine *MessageBox* mit einer Fehlermeldung geöffnet.

4.3.3.4.2 Raumsensoradresse

Die Adresse des Raumsensors, der mit dem Thermostat verbunden werden soll, kann hier eingetragen werden. Die Adresse muss eine vierstellige Hexadezimalzahl sein, andernfalls wird eine Fehlermeldung ausgegeben. Die Eingabe ist optional.

4.3.3.4.3 Thermostatadresse

Die Eingabe der Thermostatadresse ist nur möglich, wenn der *Dialog* von den Entwickleroptionen geöffnet wurde. Andernfalls ist die automatisch generierte Adresse vom Nutzer unveränderbar.

4.3.3.4.4 „Speichern“ Knopf

Der „Speichern“ Knopf übergibt die eingetragenen Werte der *AddThermostat()* Methode der *HeizungsregelungView*, die wiederrum prüft, ob schon Thermostate mit dem Name oder der Adresse vorhanden sind. Ist dies der Fall, öffnet die *AddThermostat()* Methode eine *MessageBox* mit der Frage, ob dieses überschrieben werden soll. Wenn der Nutzer verneint, bleibt der *AddThermostatDialog* geöffnet, ansonsten wird er geschlossen. Beim Schließen des Dialogs wird außerdem automatisch die eingetragene Adresse mittels des *Services changeID()* an die Adresse geschickt, die dem *AddThermostatDialog* im Konstruktor als Empfängeradresse übergeben wurde. Dadurch ändert das Thermostat seine Adresse auf die im Feld Thermostatadresse eingetragene.

4.3.3.4.5 „Abbrechen“ Knopf

Der „Abbrechen“ Knopf tut genau das, was man von ihm erwarten würde: Er schließt das Fenster, ohne irgendeine weitere Funktion auszuführen.

Damit ist die Beschreibung des *AddThermostatDialogs* abgeschlossen. Im nächsten Kapitel erfolgt die Beschreibung der Entwickleroptionen, die im *DeveloperDialog* untergebracht sind.

4.3.3.5 DeveloperDialog

Der *DeveloperDialog* beinhaltet alle Einstellungen, die dem normalen Nutzer nicht zur Verfügung stehen sollen. Er wird geöffnet, wenn in der *HeizungsregelungView* Klasse ein Rechtsklick auf die Statusanzeige des Thermostats erfolgt. Folgende Aktionen können von ihm gestartet werden:

- Den Raumnamen, die Raumsensoradresse oder die Geräteadresse (nur lokal) eines Thermostats ändern
- Ein Thermostat löschen
- Dem Thermostat eine „gemessene“ Temperatur vorgeben
- Den Motor eines Thermostats justieren
- Die Adresse eines beliebigen Thermostats ändern (verschickt einen Änderungsbefehl)
- Eine Push-Notification an die *WieDAS-App* verschicken

Das Klassendiagramm ist in Abbildung 31 zu sehen:

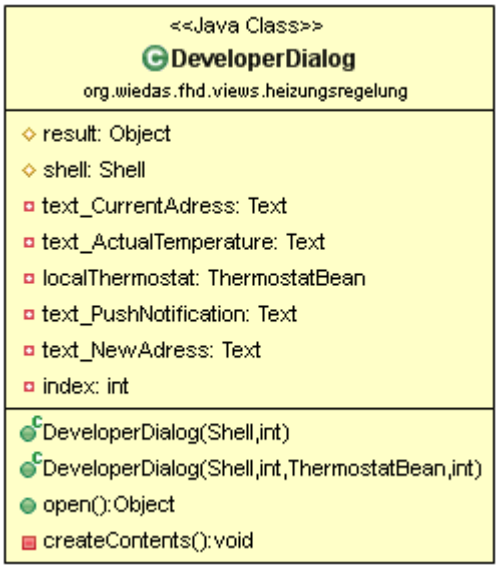


Abbildung 31: *DeveloperDialog* Klasse

Die Attribute und Methoden sind im Großen und Ganzen die Gleichen wie beim *AddThermostatDialog*, weshalb sie nicht noch einmal explizit vorgestellt werden.

Die entscheidende Logik versteckt sich in den *EventHandlern* der Benutzersteuerelemente, welche im folgenden Kapitel vorgestellt werden.

4.3.3.6 DeveloperDialog – Benutzeroberfläche

Der *DeveloperDialog* hat keine eigene komplexe Logik hinterlegt, er greift stattdessen auf die Methoden der *HeizungsregelungView* Klasse und die *Services* des *DeviceControllers* zu. Die realisierte Benutzeroberfläche ist in Abbildung 32 zu sehen.

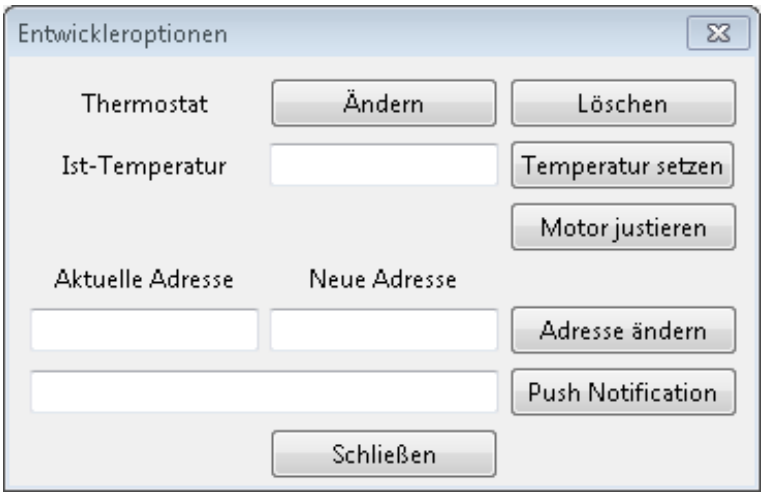


Abbildung 32: *DeveloperDialog GUI*

Der *DeveloperDialog* bekommt in seinem Konstruktor, sofern vorhanden, eine Kopie des aktuell ausgewählten Thermostats übergeben. Sollte kein Thermostat vorhanden sein, sind die Bedienelemente in den oberen drei Zeilen deaktiviert.

4.3.3.6.1 Ändern

Durch Drücken des „Ändern“ Knopfes wird eine Instanz des *AddThermostatDialogs* geöffnet. Werden in dieser Instanz Änderungen am übergebenen Thermostat vorgenommen, werden sie in das Thermostat der *HeizungsregelungView*, der *DeviceController* Klasse und in die Datenbank übertragen. Das Thermostat in der Datenbank wird allerdings dadurch überschrieben, sodass die bis dahin aufgezeichneten Temperaturen verloren gehen.

4.3.3.6.2 Löschen

Wird der „Löschen“ Knopfes gedrückt, öffnet sich eine *MessageBox* und fordert eine Bestätigung, dass das Thermostat wirklich gelöscht werden soll. Wenn diese gegeben wird, wird das aktuell in der *HeizungsregelungView* ausgewählte Thermostat sowohl aus sämtlichen Listen, als auch aus der Datenbank gelöscht. Dies geschieht durch die Methode removeThermostat der *HeizungsregelungView* Klasse.

4.3.3.6.3 Temperatur setzen

Möchte man dem aktuell ausgewählten Thermostat, z.B. zu Testzwecken, eine „gemessene“ Temperatur vorgeben, so kann man diese in das Textfeld neben dem Schriftzug „Ist-Temperatur“ eintragen und auf den „Temperatur setzen“ Knopf drücken. Daraufhin übernimmt das Thermostat diese als ob es sie selbst gemessen hätte. Die Funktion wird durch den *setActualTemperature Service* des *DeviceControllers* umgesetzt, durch den auch die Raumsensortemperatur übertragen wird.

4.3.3.6.4 Motor justieren

Um den Motor des aktuell ausgewählten Thermostats zu justieren, kann man auf den „Motor justieren“ Knopf drücken. Dadurch verfährt den Motor von Endanschlag zu Endanschlag und zählt auf dem Weg die Inkremente der Lichtschranke, mit deren Hilfe die Heizungsregelung seine aktuelle Position berechnet. Die Funktion wird durch den *motorAdjust Service* des *DeviceControllers* umgesetzt.

4.3.3.6.5 Adresse ändern

Mit dem „Adresse ändern“ Knopf wird die Geräteadresse, die im linken Textfeld eingetragen ist von dem jeweiligen Thermostat auf die im rechten Textfeld eingetragene geändert. Dies geschieht unab-

hängig vom in der *HeizungsregelungView* ausgewählten Thermostat. Die Funktion wird durch den *changeID Service* des *DeviceControllers* umgesetzt.

4.3.3.6.6 Push Notification

Dieser Knopf sendet mithilfe des von Andreas Knoll während seiner Bachelorthesis umgesetzten *Push Notification Services* eine Nachricht an die *WieDAS Android-App* (Knoll, 2013). Die zu sendende Nachricht muss dabei im nebenstehenden Textfeld eingetragen werden. Diese Schaltfläche wurde zu Testzwecken für den entsprechenden *Service* eingefügt.

Das nächste Kapitel stellt die *TemperatureDrawDialog* Klasse vor, welche den Temperaturverlauf über der Zeit anzeigen kann.

4.3.3.7 TemperatureDrawDialog

Der *TemperatureDrawDialog* dient der Darstellung der gemessenen Temperaturen eines Thermostats über der Zeit. Er hat als Übergabeparameter im Konstruktor eine Instanz der Klasse *ThermostatBean*, für die dann der Verlauf gezeichnet wird. Das Klassendiagramm ist in der folgenden Abbildung 33 zu sehen:

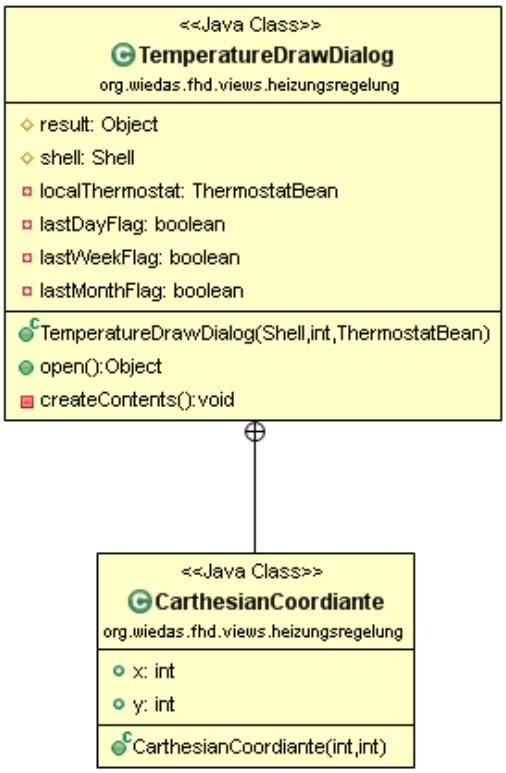


Abbildung 33: *TemperatureDrawDialog* Klasse

Zunächst werden die Attribute und Methoden der Klasse vorgestellt, danach wird erläutert, wie man mithilfe der von *SWT* bereitgestellten Mittel dynamisch Zeichnen kann und anschließend wird die Benutzeroberfläche mit einem beispielhaft gezeichneten Resultat gezeigt.

4.3.3.7.1 result

Das *result* wird beim Schließen des Shells (also des Fensters) als Ergebnis der Methode *open()* zurückgegeben, welche zur Klasse *Dialog* gehört und hier erweitert wird. Es wird in diesem Fall nicht weiter verarbeitet, weshalb auch nicht genauer darauf eingegangen wird.

4.3.3.7.2 Shell

Das *Shell* repräsentiert bei *SWT* ein Fenster, welches dann mit Benutzersteuerelementen bestückt werden kann. Es wird von der Methode *open()* des *Dialogs* geöffnet.

4.3.3.7.3 localThermostatBean

Die lokale Kopie des im Konstruktor übergebenen Thermostats. Die aufgezeichneten Temperaturen dieses Thermostats werden genutzt, um das Koordinatensystem und den Verlauf zu zeichnen.

4.3.3.7.4 lastDayFlag, lastWeekFlag & lastMonthFlag

Diese drei Variablen vom Typ *boolean* sind *Flags* die die aktuelle Auswahl des Nutzers bzgl. der Zeitachse festhalten. Weil das Zeichnen in einem anderen *EventHandler* erfolgt (darauf wird später noch eingegangen), als die Reaktion auf das Klicken der Buttons für die Auswahl, wurde der Umweg über diese Variablen gewählt. Wird ein Knopf vom Nutzer gedrückt, wird das entsprechende *Flag* gesetzt und das Event zum Zeichnen ausgelöst.

4.3.3.7.5 Konstruktor TemperatureDrawDialog()

Hier wird dem *Dialog*, neben einem vom *ResourceManager* bereitgestellten *Shell*, die Instanz des Thermostats übergeben, für die dann der Temperaturverlauf gezeichnet wird.

4.3.3.7.6 open()

Öffnet das *Shell*, platziert das Layout darin und wartet solange bis es geschlossen wurde. Gibt anschließend das *result* zurück.

4.3.3.7.7 createContents()

Wie beim *Dialog* üblich, werden auch hier die Benutzersteuerelemente in dem Layout platziert, skaliert, mit Inhalt versehen und mit *EventHandlern* ausgestattet. Daher enthält sie auch hier den größten Teil der Logik. Diese wird im Verlauf der Kapitel 4.3.3.8 - Zeichnen mit SWT und 4.3.3.9 - TemperatureDrawDialog – Benutzeroberfläche näher erklärt.

4.3.3.7.8 CarthesianCoordinate Klasse

Diese Klasse dient der Speicherung einer x- und einer y-Koordinate. Sie wird als Liste verwendet, die dazu genutzt wird, um die Koordinaten der zu zeichnenden Punkte übersichtlicher zu speichern.

Im nun folgenden Kapitel wird kurz erläutert, wie man zur Laufzeit der Anwendung mithilfe von *SWT* zeichnen kann.

4.3.3.8 Zeichnen mit SWT

Möchte man zur Laufzeit mithilfe von *SWT* zeichnen, gibt es zwei Möglichkeiten. Soll auf einem *Image* gemalt werden, muss man die Klasse *org.Eclipse.SWT.graphics.GC* benutzen. Soll hingegen auf einem *Control*[12] gezeichnet werden, muss man das *paintEvent* nutzen. Dieses greift letzten Endes allerdings auch auf die GC Klasse zu. *SWT* bietet ein spezielles *Control* zum Zeichnen, *Canvas*, welches einige Einstellmöglichkeiten für das Zeichnen bietet. Dieses wird im weiteren Verlauf eingesetzt, weshalb die Erläuterung für das *paintEvent* erfolgt.

Das Einbinden eines *Canvas* in eine Benutzeroberfläche wird im folgenden Code-Beispiel gezeigt.

[12] Ein *Control* ist ein Benutzersteuerelement, z.B. ein Button, ein Textfeld oder eine *ComboBox*.

```
1   final Canvas canvas = new Canvas(shell,SWT.NONE);
2   canvas.setLocation(0, 0);
3   canvas.setSize(100, 100);
4   canvas.setBackground(ResourceManager.getColor(SWT.COLOR_WHITE));
5   canvas.addPaintListener(new PaintListener(){
6       public void paintControl (PaintEvent e){
7       //Hier erfolgt das Zeichnen
8       }
9   });
```

Listing 10: Zeichnen mittels der Klasse *Canvas*

Wichtig: Der Koordinatenursprung liegt in der oberen linken Ecke des Bildschirms.

In den Zeilen 1-4 wird das *Canvas* zuerst angelegt, positioniert, skaliert und anschließend mit der Hintergrundfarbe Weiß versehen. Der Ursprung des *Canvas* liegt dabei in der linken oberen Ecke des Fensters und der Endpunkt liegt bei 100 Pixel nach unten und 100 Pixel nach rechts. Die Farben können entweder über die Klasse *Display* oder, wie in diesem Fall, über den *ResourceManager* mit `ResourceManager.getColor()` bezogen werden.

In Zeile 5 wird schließlich der *PaintListener* definiert und in Zeile 7 kann man den Code zum Zeichnen ausführen.

Das *paintEvent* wird beim Öffnen des Dialogs einmal automatisch ausgeführt und außerdem bei jeder neuen Positionierung des Shells, zu dem das *Canvas* gehört (z.B. Verschieben des Fensters). Man kann es auch manuell auslösen, in dem man die beiden Methoden

```
Canvas.redraw();
Canvas.update();
```

ausführt. Diese stammen von der Klasse *Control*, wobei die erste den kompletten Bereich des *Controls* als „muss neu gezeichnet werden“ markiert und die zweite alle ausstehenden Zeichnen-Operationen ausführt.

Anhand eines weiteren Beispiels wird das Zeichnen einer Linie von der linken oberen Ecke in die rechte untere Ecke gezeigt:

```
1  e.gc.setLineWidth(3);
2  e.gc.setLineStyle(SWT.LINE_SOLID);
3  e.gc.setForeground(ResourceManager.getColor(SWT.COLOR_BLACK));
4  e.gc.drawLine(0,0,canvas.getClientArea().width,canvas.getClientArea().height);
5  e.gc.setForeground(ResourceManager.getColor(SWT.COLOR_RED));
6  e.gc.drawText("Hello World!", 50, 50);
```

Listing 11: Zeichnen einer Linie und eines Textfeldes

Die Zeilen 1-3 enthalten Einstellungen für das in Zeile 4 erfolgende Zeichnen. In Zeile 1 wird die Breite der Linie auf 3 Pixel gesetzt, in Zeile 2 wird das Aussehen der Linie auf „*solid*“ gesetzt (es sind auch Punkte oder alternierend Punkt-Strich möglich) und in Zeile 3 wird die Farbe auf schwarz gestellt. Wichtig hierbei ist, dass es nur zwei Farbeinstellungen gibt, entweder Hintergrundfarbe oder Vordergrundfarbe. Die Vordergrundfarbe gilt dadurch auch für Texte in Textfeldern und alle anderen Linien.

In Zeile 4 erfolgt dann das Zeichnen, wobei zuerst die x- und y-Koordinate des Ursprungs und danach die des Ziels angegeben werden. Mittels `Canvas.getClientArea()` kann man auf die Eigenschaften des Zeichenbereichs zugreifen, hier z.B. auf die Breite und Höhe.

In Zeile 5 wird die Vordergrundfarbe für den folgenden Text auf Rot gestellt, welcher in Zeile 5 an der Position 50 Pixel nach unten und 50 Pixel nach rechts mit dem Inhalt „Hello World!“ ausgegeben wird.

Es gibt noch viele weitere Möglichkeiten die Darstellung zu beeinflussen, z.B. Linienabschluss, Textgröße, Schriftart, usw., und viele weitere Formen außer Linien und Textfeldern, welche hier aber nicht explizit aufgeführt werden.

Mittels dieser Methoden wurde die im nächsten Kapitel vorgestellte Benutzeroberfläche realisiert.

4.3.3.9 TemperatureDrawDialog – Benutzeroberfläche

Die Benutzeroberfläche besteht aus einer *Canvas*, unter welcher sich drei Knöpfe für die Auswahl der verschiedenen Zeitspannen befinden. Das fertige Resultat, inklusive einer Darstellung des Temperaturverlaufs innerhalb von vierundzwanzig Stunden, ist in Abbildung 34 zu sehen:

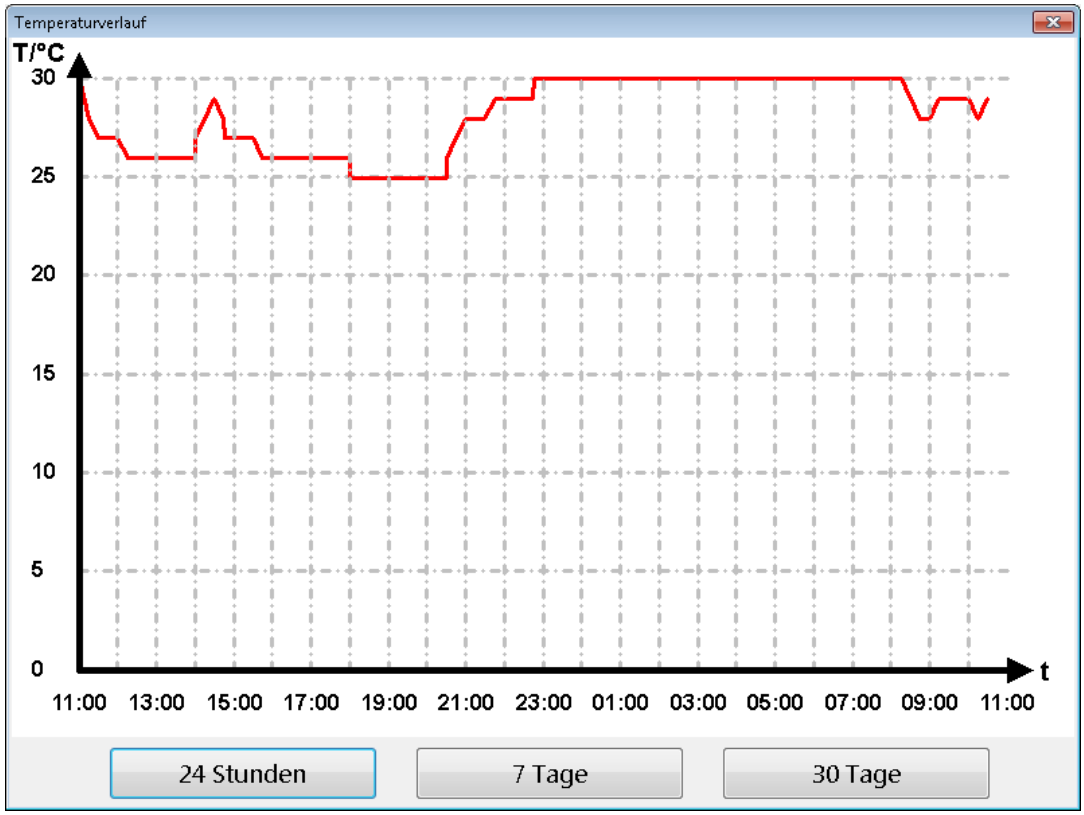
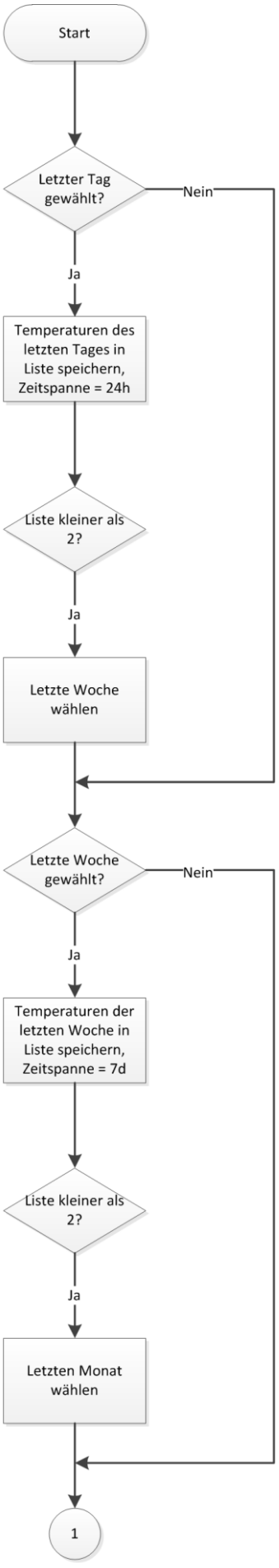
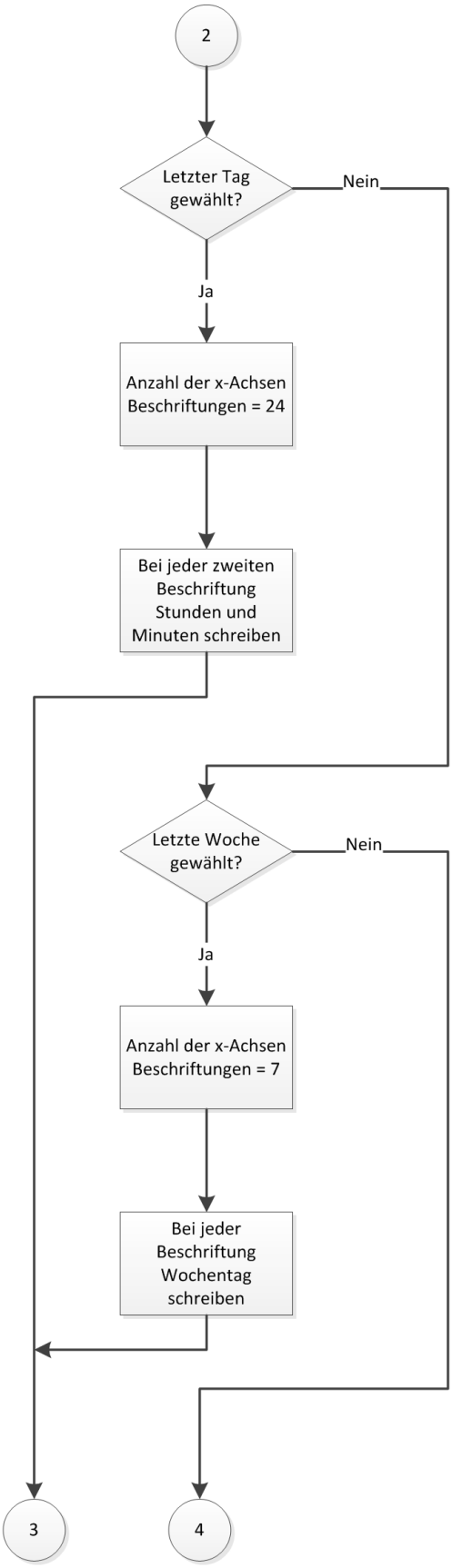


Abbildung 34: *TemperatureDrawDialog GUI*

Die Größe des Fensters beträgt 800x600 Pixel, weil diese Auflösung auch auf älteren Bildschirmen zur Verfügung stehen sollte. Die Größe kann allerdings im Editor angepasst werden, ohne die Darstellung durcheinander zu bringen, die Zeichnung erfolgt relativ zur Fenstergröße. Bei einer zu kleinen Fenstergröße würde lediglich der Text so nahe zusammen rücken, dass er nicht mehr lesbar wäre.

Weil die Achsen inklusive Beschriftung je nach ausgewählter Zeitskala und je nach Temperaturspanne skaliert werden, muss im *EventHandler* des *paintEvents* noch eine Auswertung erfolgen. Diese ist in dem folgenden Programmablaufplan zu sehen.


Start
Letzter Tag gewählt?
Nein
Ja
Temperaturen des letzten Tages in Liste speichern, Zeitspanne = 24h
Liste kleiner als 2?
Ja
Letzte Woche wählen
Letzte Woche gewählt?
Nein
Ja
Temperaturen der letzten Woche in Liste speichern, Zeitspanne = 7d
Liste kleiner als 2?
Ja
Letzten Monat wählen
1


2
Letzter Tag gewählt?
Nein
Ja
Anzahl der x-Achsen Beschriftungen = 24
Bei jeder zweiten Beschriftung Stunden und Minuten schreiben
Letzte Woche gewählt?
Nein
Ja
Anzahl der x-Achsen Beschriftungen = 7
Bei jeder Beschriftung Wochentag schreiben
3
4

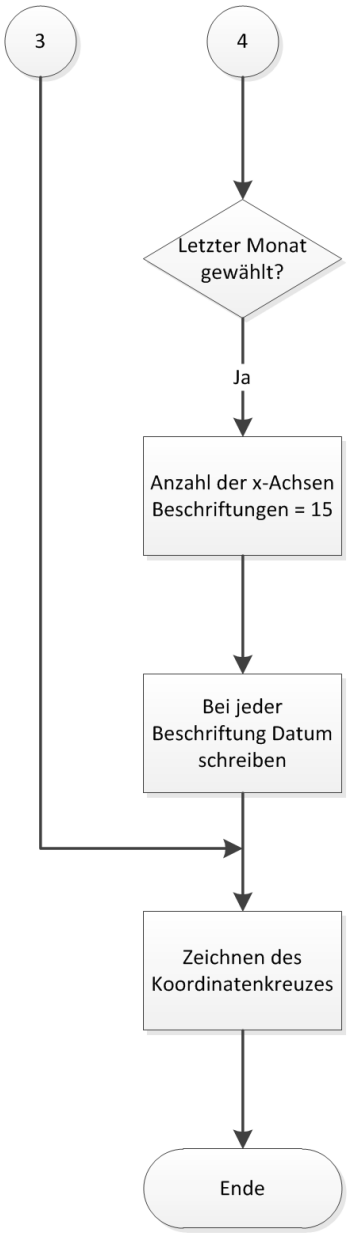


Abbildung 35: PAP Zeichnen des Temperaturverlaufs

Durch diesen Programmablauf wird zuerst sichergestellt, dass auch genug Werte für eine Darstellung zur Verfügung stehen. Sind weniger als zwei Werte in der Liste der zu zeichnenden Werte vorhanden, wird auf die nächste Zeitskala umgeschaltet. Das Öffnen des Dialogs wird von der *Heizungsregelung-View* Klasse nur ermöglicht, wenn insgesamt mindestens zwei Werte für das Thermostat vorhanden sind, weshalb diese spätestens in der „30 Tage“ Ansicht zu finden sein müssen. Alle älteren Werte werden vom jeweiligen *ThermostatBean* automatisch gelöscht.[13]

Im nächsten Kapitel wird abschließend noch auf die *OSGi ServiceBinder* des *View Bundles* eingegangen.

[13] Vgl. Kapitel 4.3.1.4.12 - addToTemperatureStorage

4.3.3.10 Die View-ServiceBinder

Die *ServiceBinder* im *View Bundles* haben die gleiche Aufgabe wie die des *Device Bundles*, nämlich *getter* und *setter* Methoden für das vom Framework übergebene Interface zur Verfügung zu stellen. Ihre Klassendiagramme sind in Abbildung 36 dargestellt.

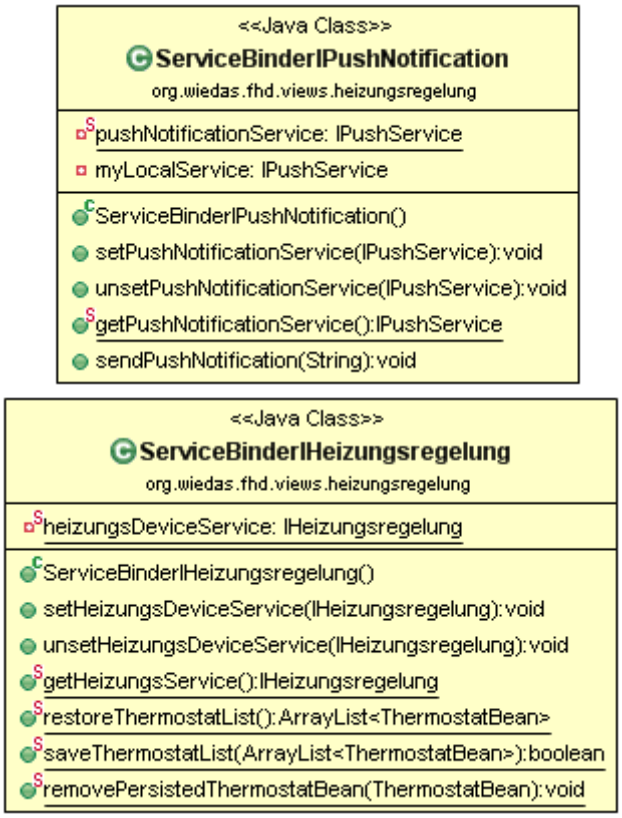


Abbildung 36: *ServiceBinder* Klassen in der *View*

Auf den genauen Zusammenhang der *ServiceBinder* mit dem *OSGi*-Framework wird in diesem Kapitel nicht eingegangen, er kann im Kapitel 4.3.1.3 - Die Device-ServiceBinder nachgelesen werden.

Die jeweiligen *set-* / *unset*-Methoden dienen dem Einbinden der Interface-Instanzen, die sie vom *OSGi*-Framework übergeben bekommen. Alle anderen Methoden stellen lediglich die gleichnamigen *Services* zur Verfügung, damit man bei einem Zugriff auf diese nicht jedes Mal die Methode *getHeizungsService()* bzw. *getPushNotificationService*() ausführen muss.

Im nächsten Kapitel folgt eine kurze Beschreibung der Zusammenhänge bei der Anmeldung von neuen Thermostaten im System.

4.3.4 Die Anmeldung neuer Thermostate

Der Ablauf bei Anmeldung eines neuen Thermostats im System ist ein Zusammenspiel mehrerer Klassen und *Bundles*, welches zwar in den vorherigen Kapiteln teilweise beschrieben wurde, aber im Gesamtkontext wesentlich besser verständlich ist. Deshalb wird hier noch einmal kurz beschrieben, wie der Vorgang genau abläuft. Am deutlichsten wird es anhand einer Abbildung:

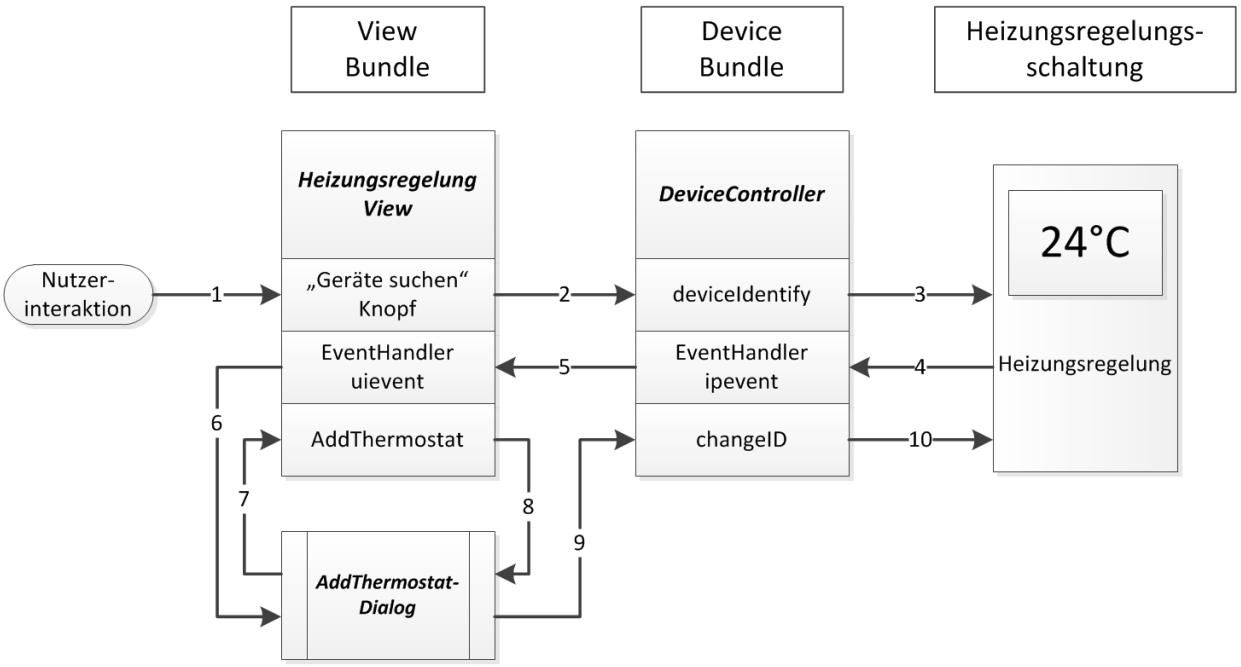


Abbildung 37: Ablauf der Anmeldung neuer Thermostate

Der Vorgang ist visuell nach Bundlezugehörigkeit geordnet. Links befindet sich das *View Bundle*, rechts daneben das *Device Bundle* und ganz rechts ist eine symbolische Darstellung der Heizungsregelungsschaltung. Beim *View Bundle* sind zwei Klassen beteiligt, die *HeizungsregelungView* und der *AddThermostatDialog*, beim *Device Bundle* nur der *DeviceController* (abgesehen vom *ThermostatBean*). Die kleineren Kästchen unter dem kursiv geschriebenen Klassennamen symbolisieren die Methoden der Klassen.

Ausgelöst wird die Suche nach Thermostaten durch einen Linksklick des Nutzers auf den „Geräte suchen“ Knopf der *HeizungsregelungView* (1). Daraufhin wird der *Service deviceIdentify()* des *DeviceControllers* aufgerufen (2), welcher eine Anfrage zur Identifikation an alle Thermostate sendet (3). Diese melden sich bei Empfang zurück (4), was vom *EventHandler* des *ipevent*s empfangen wird. Dieser leitet die Identifikation als *uievent* an die *HeizungsregelungView* weiter (5), welche anschließend überprüft, ob ihr die Adresse des Thermostats bekannt ist. Wenn nicht, wird ein neuer *AddThermostatDialog* geöffnet (6), in welchem der Nutzer den Raumnamen und (optional) die Raumsensoradresse einstellen kann. Beim Drücken auf den „Speichern“ Knopf des Dialogs wird die Methode AddThermostat der *HeizungsregelungView* aufgerufen (7), die überprüft ob der Raumname schon vergeben ist. Ist dies der Fall, wird ein *Dialog* geöffnet, der den Nutzer fragt, ob er das Thermostat überschreiben will. Wenn das Thermostat unbekannt war oder überschrieben wurde, speichert sie das Thermostat in der Liste und meldet dies per Rückgabewert an den *AddThermostatDialog* (8), welcher dann den *Service changeID* mit der ermittelten Adresse aufruft (9).[14] Dieser sendet dann eine Adressänderungsaufforderung an das Thermostat.

Damit ist die Beschreibung der Umsetzung der Bachelorthesis abgeschlossen. Im nächsten Kapitel werden die Probleme, die währenddessen auftraten geschildert und die entsprechenden Lösungen vorgestellt.

[14] Der Ablauf der Adressermittlung ist in Abbildung 27 im Kapitel 4.3.3.1 - HeizungsregelungView dargestellt.

5 Probleme & Lösungen

Während der Bachelorthesis traten häufiger Probleme auf, die nicht auf Anhieb gelöst werden konnten. Ein paar davon sollen nun vorgestellt werden:

5.1 Contiki-Events

Gegen Ende des Praxisprojekts kam es vor, dass nach ein paar Minuten im Betrieb manche *Protothreads* von *Contiki* nicht mehr ausgeführt wurden, so z.B. der Regler-*Protothread*. Diese *Protothreads* wurden und werden alle vom *Main-Protothread* gesteuert, indem sie passiv auf Events warten, die ihnen vom *Main-Protothread* zugeschickt werden. Beim Empfang lesen sie gegebenenfalls die mitgeschickten Daten aus und führen ihr Programm einmal aus. Anschließend warten sie auf das nächste Event. Zur Lösung dieses Fehlers wurden die Debugging-Möglichkeiten von Contiki verwendet:

Contiki ermöglicht das Debugging durch die Ausgabe von Meldungen über die *USART* – Schnittstelle des Mikrocontrollers, dafür muss lediglich die Zeile `#define DEBUG DEBUG_PRINT;` im Programm vorhanden sein. Danach gibt *Contiki* alle sechzig Sekunden eine Statusmeldung heraus und zeigt beim Systemstart einmalig den Speicherbedarf der im Flashspeicher hinterlegten Daten an. Zusätzlich können mit dem Befehl `printf("");` eigene Meldungen ausgegeben werden, die z.B. den Zustand bestimmter Variablen beinhalten.

Durch die Nutzung dieser Möglichkeiten stellte sich heraus, dass das Problem der „einfrierenden" *Protothreads* dadurch verursacht wurde, dass manche Events nicht vom System zugestellt werden konnten und dadurch die *Eventqueue* ausgelastet wurde[15]. Nach einer Weile konnten dadurch keine Events mehr zugestellt werden und das System „fror ein". Die Ursache war, dass im *Main-Protothread*, dem zentralen Verwaltungsprotothread für ankommende Daten, mit dem Befehl PROCESS_WAIT_EVENT() auf eintreffende Events gewartet wurde, um anschließend auf sie zu reagieren. Durch diesen Befehl hält der *Protothread* an, bis ein beliebiges Event eintrifft und wird danach vom System fortgesetzt. Im Gegensatz dazu gibt es noch den Befehl PROCESS_WAIT_EVENT_UNTIL(), welchem man zusätzlich Bedingungen mitgeben kann, z.B. ein bestimmtes Event oder einen abgelaufenen Timer.

Weil festgestellt wurde, dass die Events zwar verschickt worden waren, aber nicht ankamen und gleichzeitig die *Eventqueue* volllief, wurde die Ursache im PROCESS_WAIT_EVENT() Befehl vermutet. Nach der Änderung des Befehls auf PROCESS_WAIT_EVENT_UNTIL() mit der Übergabe der verschiedenen Events als Bedingung, stauten sich die Events nicht mehr in der *Eventqueue* und konnten empfangen werden. Warum dies der Fall ist konnte leider nicht nachvollzogen werden, da die Dokumentation darüber keine Auskünfte gibt und die Contiki-Routinen, die diesen Befehl umsetzen, nur bereits kompiliert erhältlich waren.

5.2 Motortreiber

Die Inbetriebnahme der Hardware nach der Fertigstellung verlief größtenteils problemlos, allerdings war die Stromaufnahme der gesamten Schaltung gegenüber der Vorgängerversion stark erhöht, von ~20mA auf ~70mA. Da die größte Änderung gegenüber der vorherigen Schaltung der Motortreiber war, wurde die Ursache hier vermutet. Die Ruhestromaufnahme des Motortreibers beträgt laut Datenblatt bis zu 48mA (SGS-THOMSON Microelectronics, 1996), was ebenfalls auf diesen als Ursache

[15] Nebenbei wurde dabei die maximale Anzahl von 32 Events in der Eventqueue ermittelt. Diese konnte sonst keiner Dokumentation entnommen werden.

hinwies. Durch Spannungsmessungen konnte schließlich ermittelt werden, dass sich, weil der *Enable*-Eingang des ICs im Softwaretreiber des Mikrocontrollers nicht permanent auf „*Low*“ gesetzt wurde, ein schwebendes Potential ergab, das ausreichte den Treiberbaustein zu schalten und den Ruhestrom fließen zu lassen. Nach der Anpassung der Motortreiber, sodass nun permanent „*Low*“ am *Enable*-Eingang des Treibers anliegt, sank die Ruhestromaufnahme auf ~35mA. Trotzdem wurde für die nächste Hardwarerevision eine weitere Anpassung vorgenommen, sodass jetzt ein NPN-Transistor, der ebenfalls am *Enable*-Signal des Mikrocontrollers liegt, in der Spannungsversorgung des Motortreibers hängt. Laut Datenblatt kann der Transistor bis zu 800mA schalten, was dafür ausreichend ist. Dadurch kann der Treiberbaustein sicher abgeschaltet werden und die Ruhestromaufnahme der Schaltung sollte weiter reduziert werden.

5.3 EEPROM

Während der Erstellung der *GUI* der Heizungsregelung mussten mehrere Grafiken „von Hand“ erstellt werden, wie in Kapitel 4.2.1 - Die grafische Benutzeroberfläche beschrieben. Der Speicherbedarf der Grafiken betrug ca. 1280 Byte, was den Arbeitsspeicher des Mikrocontrollers schnell füllte. So wurden nach dem Einspielen der Software plötzlich keine Debugginginformationen über die *USART* Schnittstelle mehr ausgegeben. Abhilfe schaffte hier eine Auslagerung der Grafiken in das *EEPROM* des Mikrocontrollers mittels der von Atmel bereitgestellten Treiberbibliotheken.

Das Speichern der Grafiken ins *EEPROM* erfolgt durch den Compiler bei der Initialisierung der Variablen. Möchte man zur Laufzeit des Programms schreibend auf eine Variable zugreifen, kann man die von Atmel bereitgestellten Funktionen *eeprom_write_byte* und *eeprom_write_block* benutzen. Im Falle der Heizungsregelung wurden sie genutzt, um die Geräteadresse bei einer Änderungsaufforderung im *EEPROM* zu hinterlegen, damit sie bei einem Neustart wieder zur Verfügung steht. Dies führte allerdings zu einem Fehler, da nicht bedacht wurde, dass das erste geschriebene Byte bei einem Zugriff auf das *EEPROM* fehlerhaft sein kann. Die Lösung ist das Schreiben eines „Dummy“ Bytes, also einem Byte ohne tieferen Sinn, bevor die tatsächliche Geräteadresse gespeichert wird.

5.4 Funkverbindung

Ein sehr interessanter Fehler trat vor der Realisierung der neuen Hardware auf. Bei dem Versuch, die Funkverbindung mit der alten Hardwareversion aus dem Praxisprojekt herzustellen, konnten nur sporadisch Daten empfangen werden, obwohl das Programm auf anderen Schaltungen fehlerfrei funktionierte. Deshalb konnte ein Softwarefehler ausgeschlossen werden. Wenn Daten empfangen werden konnten, hatte die Schaltung allerdings eine um ~ 120mA erhöhte Stromaufnahme. Die Ursache war schließlich Folgendes:

Die Referenzspannung des A/D-Wandlers darf beim *ATmega128RFA1* nicht höher als 1,8V sein, dies gilt ebenso für eine externe Referenzspannung. Angeschlossen war aber die Versorgungsspannung des Controllers von 3,3V. Durch das Verwenden der internen Spannung von 1,8V und den Anschluss eines 100nF Kondensators an den Referenzspannungs-Pin zur Stabilisierung konnten dann permanent Daten empfangen werden, auch ohne hohe Stromaufnahme Die Ursache für das sporadische Funktionieren konnte nicht ermittelt werden, da der Mikrocontroller quasi eine „Black Box“ ist und keine internen Schaltpläne erhältlich sind. Fraglich bleibt auch, warum der Mikrocontroller durch eine so hohe Stromaufnahme keinen merklichen Schaden nahm.

Im folgenden Kapitel wird ein Ausblick auf die möglichen Erweiterungen der Arbeit gegeben.

6 Ausblick

Die in der Zielsetzung genannten Punkte konnten zwar umgesetzt werden, dennoch wären noch Verbesserungen möglich, die aufgrund der beschränkten Zeit der Bachelorthesis nicht mehr realisierbar waren:

- Die Optimierung der Regelung. Momentan funktioniert die Regelung zwar, d.h. bei einer höheren Temperatur als der gewünschten fährt sie das Ventil schrittweise zu und umgekehrt, aber über die Genauigkeit und Stabilität der Regelung liegen noch keine Daten vor. Momentan ist die Zykluszeit der Regelung in der Datei *controller.h* auf 150 Sekunden eingestellt. Die Auswirkungen einer längeren / kürzeren Zykluszeit auf die Stabilität könnten ebenfalls untersucht werden.

- Die Sicherung der Kommunikation. Eine weitere Optimierungsmöglichkeit wäre einerseits die Verwendung einer verschlüsselten Kommunikation der Heizungsregelung mit dem *OSGi*-Framework. Dazu bietet der Mikrocontroller bereits einen Hardwareverschlüsselungschip, der in der Lage ist Daten mit dem *AES128* Verfahren zu verschlüsseln. Andererseits wäre, um den Verlust von Funkbefehlen zu vermeiden, entweder die Verwendung eines verbindungsorientierten Kommunikationsprotokolls wie TCP zu Überprüfen oder eine Bestätigung der Heizungsregelung für jeden Befehl zu implementieren, ähnlich der Struktur der im Kapitel 4.3.1.2.3 - *changeID* & *IdChangeThread* beschrieben Adressübertragung.

- Das Auslagern der Logik aus dem *View* in das *Device Bundle*. Laut der im Labor eingesetzten Bundlestruktur sollte die gesamte Logik der Anwendung bei den *Device Bundles* des jeweiligen Gerätes liegen. Dadurch kann eine beliebige Benutzeroberfläche auf die Logik zugreifen und diese muss nicht angepasst werden. So könnte die Adressvergabe, die derzeit noch in dem *View Bundle* stattfindet, in das *Device* ausgelagert werden.

- Die Energieoptimierung. So könnten beispielsweise die Energiesparmodi des Mikrocontrollers eingebunden werden und die Schaltung mit dem neu entworfenen Layout, in dem der hohe Ruhestromverbrauch des Motortreibers eingeschränkt ist, aufgebaut werden. Außerdem könnte die aufgenommene Leistung den einzelnen Komponenten zugeordnet werden und dementsprechend Maßnahmen zur Reduktion getroffen werden.

- Zuletzt könnte man die Heizungsregelungsschaltung noch verkleinern, indem man den Konnektor des Touch Panels auf die Oberseite der Platine verlegt und den Raum unter dem Display besser ausnutzt. Dadurch könnte die fertige Schaltung eventuell in das Gehäuse des Thermostats passen, aus dem der Motor entnommen wurde.

Zum Abschluss der Bachelorthesis folgt noch ein kurzes Fazit.

7 Fazit

Die Heizungsregelung ist nun in der Lage per Funk Befehle zu erhalten und kann mit dem Touch Panel und einer dafür realisierten Benutzeroberfläche bedient werden. Es wurde ebenfalls eine Anwendung für die *WieDAS* Anwendung, welche mit dem *OSGi*-Framework läuft, realisiert.

Diese beinhaltet ein logisches Gerät, das *Device*, welches eine Liste aller bekannten Thermostate beinhaltet und diese in einer Datenbank persistiert. Das *Device* ermöglicht zusätzlich die Kommunikation mit dem Thermostat und exportiert diese als *Services*. Zusätzlich ist auch eine grafische Benutzeroberfläche vorhanden, die *View*, welche dem Nutzer die Interaktion mit dem Thermostat ermöglicht und außerdem eine automatische Adressvergabe und einen Dialog zum Zeichnen des Temperaturverlaufs für die Thermostate beinhaltet.

Aufgrund der begrenzten Zeit der Bachelorarbeit konnte ich mich nicht in aller Tiefe in die *OSGi*-Strukturen und das *JPA* Konzept vertiefen, aber das erarbeitete Wissen reichte aus, um mir einen Überblick über die Möglichkeiten zu geben und die gewünschten Ziele umzusetzen. Im Nachhinein finde ich, dass *OSGi* ein interessantes Konzept ist, wobei in der *WieDAS* Anwendung durch die geringe Bundleanzahl pro Gerät (ein *Device* und eine *View*) die Modularisierung nicht ihre ganze Stärke entfalten kann. Außerdem bestehen weiterhin gewisse Abhängigkeiten in meinem Projekt. So muss das *View Bundle* noch die *ThermostatBean* Klasse aus dem *Device* importieren, damit die Daten von beiden *Bundles* in einer einheitlichen Struktur abgelegt werden können. Allerdings besteht diese Abhängigkeit nur vom *View Bundle* aus, welches ohne das *Device Bundle* seinen Zweck sowieso nicht erfüllen könnte.

Alles in allem war die Bachelorarbeit sehr lehrreich und ich habe viele interessante Einblicke in informationstechnische Systeme bekommen, die ich vorher im Verlauf des Studiums so nicht kennengelernt hatte.

8 Abkürzungsverzeichnis

6LoWPAN	IPv6 over Low-power Wireless Personal Area Networks	Kommunikationsprotokoll, welches →IPv6 für energieeffiziente Systeme umsetzt.
EEPROM	Electrically Erasable Programmable Read-Only Memory	Elektrisch löschbarer und programmierbarer, nichtflüchtiger Datenspeicher, der die Daten auch ohne anliegende Versorgungsspannung behält.
GUI	Graphical User Interface	Grafische Benutzerschnittstelle, ermöglicht das Bedienen und Beobachten des technischen Geräts durch den Benutzer.
ICMP	Internet Control Message Protocol	Protokoll das die Überprüfung von Teilnehmern im Netzwerk ermöglicht. So ist z.B. EchoRequest/Reply im „Ping“-Programm umgesetzt.
IDE	Integrated Development Environment	Integrierte Entwicklungsumgebung, in der viele Funktionen zur Softwareentwicklung wie Linker, Debugger und Compiler enthalten sind.
IP	Internet Protocol	Verbindungsloses Protokoll, welches auf Basis von IP-Adressen die Zustellung von Daten in einem Netzwerk ermöglicht.
IPv4	Internet Protocol Version 4	Version 4 des Internetprotokolls. Hauptunterschied zu →IPv6 ist die Adresslänge von 32Bit. Wird (noch) häufiger eingesetzt als IPv6.
IPv6	Internet Protocol Version 6	Version 6 des Internetprotokolls. Hat eine Adresslänge von 128 Bit, wird allerdings seltener eingesetzt als →IPv4.
JPA2	Java Persistence API 2.0	Java Programmierschnittstelle die den Zugriff auf Datenbanken vereinfacht.
JTAG	Joint Test Action Group	Bezeichnet den IEEE-Standard 1149.1, eine Ansammlung von Verfahren zum Debuggen von Hardware in der Schaltung.

OSGi	Open Services Gateway initiative (obsolet)	Modularisierungskonzept für Java, bei dem Softwarepakete (Bundles) und ihre Dienste (Services) dynamisch zur Laufzeit hinzugefügt werden können.
SPI	Serial Peripheral Interface	Synchroner, serieller, vollduplex-fähiger Master-Slave-Datenbus. Wird hauptsächlich zur Kommunikation mit schnelleren Komponenten auf einer Platine eingesetzt.
SWT	Standard Widget Toolkit	Programmbibliothek zur GUI-Erstellung für Eclipse, die den Zugriff auf die nativen Benutzeroberflächen des Betriebssystems ermöglicht.
TCP	Transmission Control Protocol	Verbindungssicheres Kommunikationsprotokoll, wird oft aufbauend auf→IP eingesetzt. Verbindung ist bidirektional und Datenverluste werden erkannt und behoben.
UDP	User Datagram Protocol	Verbindungsloses Protokoll, schnell weil Daten nicht auf Korrektheit oder Ankunft überprüft werden.
USART	Universal Synchronous/Asynchronous Receiver/Transmitter	Integrierte Schaltung, die über eine serielle Schnittstelle Daten verschicken und empfangen kann. Dies kann entweder synchron oder asynchron geschehen.

9 Abbildungsverzeichnis

10 Literaturverzeichnis

Abts, Dietmar. 2013. Grundkurs Java - Von den Grundlagen bis zu Datenbank- und Netzanwendungen. *eBook.* s.l. : Springer Vieweg, 2013. 7., aktualisierte Auflage.

Andreas, Daniel. 2013. Entwicklung einer über Funk angebundenen steuerbaren Heizungsregelung unter Verwendung des Betriebssystems Contiki im Umfeld des Ambient Assisted Livings. *Praxisprojekt.* 2013.

Bundesministerium für Bildung und Forschung. Ambient Assisted Living Deutschland. [Online] GmbH, VDI/VDE Innovation + Technik. [Zitat vom: 16. 07 2013.] http://www.aal-deutschland.de/.

ELECTRONIC ASSEMBLY GmbH. 2011. DOGS Grafik Serie. *Datenblatt.* 05 2011.

Gosch, Eckhard. 2006. Eckhard Gosch. *eckhard-gosch.de.* [Online] 19. 11 2006. [Zitat vom: 06. 08 2013.] http://www.eckhard-gosch.de/de/images/articles/controller/H-Bridge.gif.

Keith, Mike und Schincariol, Merrick. 2009. Pro JPA2 - Meeting the Java Persistence API. *Buch.* New York : Springer Verlag, 2009.

Knoll, Andreas. 2013. Konzeption und Realisierung einer Client-Server-Anwendung zum mobilen entfernten Zugriff auf Dienste einer Ambient-Assisted Living Service Plattform. *Bachelorthesis.* 2013.

Kriens, Peter. 2008. OSGi Alliance Blog. [Online] OSGi Alliance, 17. 06 2008. [Zitat vom: 21. 07 2013.] http://blog.osgi.org/2008/06/jsr-277-and-import-package.html.

Krüger, Markus. 2010. Entwicklung einer mikrocontrollerbasierten,sensorgesteuerten Raumluftoptimierung in der Altenpflege. *Bachelorthesis.* 2010.

SGS-THOMSON Microelectronics. 1996. L293DD - PUSH-PULL FOUR CHANNEL DRIVER WITH DIODES. *Datenblatt.* 1996.

Shelby, Zach und Bormann, Carsten. 2009. 6LoWPAN: The Wireless Embedded Internet. *Buch.* United Kingdom : Wiley, 2009.

Vogel, Lars. 2012. Vogella.com. *JPA 2.0 with EclipseLink - Tutorial.* [Online] vogella GmbH, 16. 03 2012. [Zitat vom: 16. 07 2013.] http://www.vogella.com/articles/JavaPersistenceAPI/article.html.

Wütherich, Gerd, et al. 2008. Die OSGI Service Platform - Eine Einführung mit Eclipse Equinox. *Buch.* Heidelberg : dpunkt Verlag, 2008.

11 Listingverzeichnis