

# Diploma thesis

Höhere Technische Bundeslehranstalt Leonding  
Abteilung EDV und Organisation

## Room Control System

### Home Automation

handed in by:  
GRUBMAIR David  
2011/2012  
5 BHDVK

KROPF Simon  
2011/2012  
5 BHDVK

on:  
10.05.2012

supervised and assessed by:  
Dipl. Ing. Prof. Gerald Köck  
Mag. Dr. Prof. Thomas Stütz

## **Declaration of academic honesty**

We hereby declare under penalty of perjury that the present paper was written independently and without assistance, other than the specified sources and tools used, not all the sources used verbatim or in essence taken from sites identified as such have.

Leonding, am .....

.....  
Kropf Simon

.....  
Grubmair David

## **Credits**

Our sincerest thanks to our supervisors Dipl. Ing. Prof. Gerald Köck and Mag. Dr. Prof. Thomas Stütz for assisting us in our diploma thesis and for their strong commitment.

## **Abstract**

This diploma thesis describes the development of a home automation system used to observe environmental conditions and furthermore to act based on these conditions.

The first part covers the thoughts and planning as well as the goals behind our system. Within this part we used technologies which suited best for our system. On the one side the product does not cost much and on the other side, it is user friendly and efficient. Due to the fact that we split our project up into two parts, we had to make exact interface definitions. One part includes the server-side system and the other part the microcontroller and the hardware behind it.

The second part deals with the implementation and the procedure of testing. This part is splitted up into two parts, which are the server and the microcontroller. At first the components which were developed by us and therefore the utilized technologies are described and explained and afterwards the connections between them. The critical parts, which are responsible for the business logic and the main operations, are outlined in detail.

The last part compares the planning process to the successful realization and displays the conclusions of our self evaluation.

## Contents

<b>1</b>	<b>Introduction</b>	<b>12</b>
1.1	Motivation . . . . .	12
1.2	Conceptual formulation . . . . .	12
<b>2</b>	<b>Planning</b>	<b>13</b>
2.1	Involved parties . . . . .	13
2.2	General approach . . . . .	13
2.3	Initial situation . . . . .	13
2.3.1	Quantity structure of parts . . . . .	13
2.4	Goals and requirements . . . . .	14
2.4.1	Goals of the new system . . . . .	14
2.4.2	Requirements . . . . .	14
2.5	Concept . . . . .	16
2.5.1	System concept . . . . .	16
2.5.2	Workflow description . . . . .	17
2.5.3	System architecture . . . . .	20
2.6	Project planning . . . . .	23
2.6.1	Project structure . . . . .	23
2.6.2	Time scheduling . . . . .	23
<b>3</b>	<b>Implementation</b>	<b>25</b>
3.1	Description of used technologies . . . . .	25
3.1.1	Java 6 . . . . .	25
3.1.2	JPA . . . . .	25
3.1.3	Oracle GlassFish Application Server v3.1.2 . . . . .	25
3.1.4	Apache Derby . . . . .	26
3.1.5	Arduino . . . . .	26
3.1.6	Android SDK . . . . .	28
3.1.7	ADK (Android Open Accessory Development Kit) . . . . .	30
3.2	Technical Solution . . . . .	33
3.2.1	System Architecture . . . . .	33
3.2.2	MCP (Minimal Communication Protocol) . . . . .	34
3.2.3	Data model Java EE Server . . . . .	38
3.3	Description of the system components . . . . .	43
3.3.1	Used Libraries on Arduino . . . . .	43
3.3.2	MCP Arduino . . . . .	48
3.3.3	MCP Java . . . . .	51
3.3.4	Control Apps Arduino . . . . .	59
3.3.5	Devices Java Side . . . . .	68
3.3.6	Tasks & Profiles . . . . .	73
3.3.7	SOAP Webservice . . . . .	75

3.3.8	Web Interface . . . . .	77
3.3.9	Logging Arduino . . . . .	84
3.3.10	Connection Handling . . . . .	86
3.3.11	Event Handling . . . . .	90
3.3.12	Android Application . . . . .	91
<b>4</b>	<b>Selfevaluation</b>	<b>97</b>
4.1	Personal experience . . . . .	97
4.1.1	Java EE . . . . .	97
4.1.2	Arduino . . . . .	97
4.2	Final words . . . . .	100
<b>5</b>	<b>Source directory</b>	<b>101</b>

## List of Figures

1	System package diagram . . . . .	16
2	Android ADK Connection . . . . .	17
3	Website connection . . . . .	18
4	Android Socket Connection . . . . .	19
5	SOAP Connection . . . . .	20
6	System architecture . . . . .	21
7	Project structure plan . . . . .	23
8	Android System Architecture . . . . .	29
9	System architecture . . . . .	33
10	MCPMessage Sequence Diagram . . . . .	35
11	MessageType Enum . . . . .	35
12	LogStatusType . . . . .	36
13	MCP-Activity data model . . . . .	38
14	Message data model . . . . .	39
15	Device data model . . . . .	40
16	MCP Class Diagram Arduino . . . . .	48
17	Connection Type . . . . .	50
18	MCP message overview . . . . .	51
19	DeviceMessage class . . . . .	52
20	CallMessage class . . . . .	52
21	ResultMessage class . . . . .	53
22	LogMessage class . . . . .	53
23	GlobalMessageID class . . . . .	54
24	MessageTypeEnum . . . . .	54
25	MCP communication overview . . . . .	55
26	Device Structure Arduino . . . . .	59
27	Synchronizing Devices . . . . .	62
28	Arduino Message Diagram . . . . .	63
29	Devices overview . . . . .	68
30	DeviceType class . . . . .	69
31	Device class . . . . .	69
32	AlarmSystem class . . . . .	70
33	Fan class . . . . .	70
34	Light class . . . . .	71
35	Heater class . . . . .	71
36	PowerOutlet class . . . . .	72
37	SensorDevice class . . . . .	72
38	RollerBlind class . . . . .	73
39	Profile-Task overview . . . . .	73
40	ProfileScheduler . . . . .	74
41	SOAP WebService . . . . .	76
42	Devices table . . . . .	77

43	Webinterface RollerBlind . . . . .	77
44	Webinterface Fan . . . . .	80
45	Webinterface Profile . . . . .	83
46	Webinterface add Tasks to Profile . . . . .	83
47	Webinterface Settings . . . . .	84
48	LogStatus . . . . .	85
49	Log_Message Class . . . . .	85
50	Connection Struct . . . . .	86
51	ConnectionType . . . . .	87
52	Asynchronous Behavior . . . . .	91
53	Android User Interface . . . . .	96



## List of Algorithms

1	Arduino sketch file . . . . .	27
2	Arduino starting permission intent . . . . .	31
3	Arduino BroadcastReceiver . . . . .	32
4	Arduino Bulk Transfer . . . . .	33
5	Arduino Vector Definition . . . . .	44
6	Arduino Vector Add Element . . . . .	44
7	Arduino Vector Iterator . . . . .	44
8	Arduino List Definition . . . . .	45
9	Arduino List Iterator . . . . .	45
10	Arduino Map Definition . . . . .	45
11	Arduino Map Comparer . . . . .	45
12	Arduino Map Element Access . . . . .	46
13	Arduino Map Iterator . . . . .	46
14	Arduino Queue Definition . . . . .	46
15	Arduino Queue Add Element . . . . .	46
16	Arduino Queue Dequeue Element . . . . .	46
17	Arduino Opening SdFat File . . . . .	47
18	Arduino Writing/Reading Files . . . . .	47
19	Arduino Closing Files . . . . .	47
20	Arduino MessageObserver Initialization . . . . .	49
21	MessageObserver SubscribeListener . . . . .	49
22	MessageListener OnMCPMessageReceived . . . . .	50
23	Java EE Server - MCPSocketReader . . . . .	56
24	Java EE Server - ActivityListener . . . . .	57
25	Java EE Server - MCPSocketWriter . . . . .	58
26	Arduino Control App Methods . . . . .	60
27	Arduino CallMethod . . . . .	60
28	Arduino JumpTable . . . . .	61
29	Arduino WorkQueue Add Elements . . . . .	61
30	Arduino GetProperties . . . . .	62
31	Arduino MessageType Checking . . . . .	63
32	Arduino Calling Device Methods . . . . .	64
33	Arduino IPersistable Interface . . . . .	64
34	Arduino Template Definition . . . . .	64
35	Arduino GenerateClassFromName . . . . .	65
36	Arduino Template Generation . . . . .	65
37	Arduino Calling the Load Method . . . . .	66
38	Arduino Casting IPersistable Map . . . . .	66
39	Arduino Calling the Persist Method . . . . .	66
40	Arduino ControlApp Methods . . . . .	67
41	Java EE Server - ProfileScheduler . . . . .	75
42	Java EE Server - SOAP Webservice . . . . .	76

43	Java EE Server - RollerBlind JavaScript . . . . .	78
44	Java EE Server - Webinterface RollerBlind . . . . .	79
45	Java EE Server - Fan JavaScript . . . . .	81
46	Java EE Server - Webinterface Fan . . . . .	82
47	Arduino Create Log_Message . . . . .	86
48	Arduino initSerial . . . . .	87
49	Arduino Serial receive . . . . .	88
50	Arduino Connector Callback . . . . .	88
51	Arduino Ethernet Initialization . . . . .	88
52	Arduino Retrieving Ethernet Messages . . . . .	89
53	Arduino Client Connect . . . . .	89
54	Arduino Client Send Message . . . . .	89
55	Arduino ADK Initialization . . . . .	90
56	Arduino ADK retrieving Messages . . . . .	90
57	Arduino ADK sending Messages . . . . .	90
58	Arduino GetLastConnection . . . . .	91
59	Arduino SendResult . . . . .	91
60	Android Device Reflection . . . . .	92
61	Android DeviceAddedListener . . . . .	92
62	Android ResultMessageListener . . . . .	93
63	Android Listener Subscribing . . . . .	93
64	Android SensorDevice Thread . . . . .	93
65	Android ResultMessage Receiver . . . . .	94
66	Android Retrieving ADK Messages . . . . .	95
67	Arduino Working with Strings . . . . .	98
68	Arduino Getting the Remote Address . . . . .	98
69	Arduino Int Map . . . . .	99
70	Arduino Map Problem . . . . .	99

## **List of Tables**

1	Power data Arduino . . . . .	21
2	Power data Java EE Server . . . . .	21
3	Milestones . . . . .	24

# 1 Introduction

## 1.1 Motivation

There are many home automation systems available on the market, but no software offers the ability to connect your Android tablet direct to it as the main control screen. Our goal was to make a flexible software which offers many possibilities and which is able to connect direct to an Android tablet as the central station. Many of the systems on the market are also very expensive because they use decent displays and stationary hardware. With our software, costumers can use one or more Arduino boards which are not very expensive and a decent Android hardware which the user probably already own.

Another point why we have seen the need to redesign such a system was the flexibility to use other sensors or actors. In many home automation systems users are forced to take the hardware they offer and probably have no way to connect their own, maybe cheaper hardware, if they want to. On the one hand, this means that the customer needs to develop his own drivers but on the other hand he has a maximum of flexibility.

Due to the reasons mentioned above we made a solution which is at first fully configurable by simple configuration and upgradable to the fact that devices, actors and sensors can be easily developed and inserted into the system.

## 1.2 Conceptual formulation

The concept was to write a flexible system which is upgradable and configurable by users. Therefore it has to be easily administrated and the overall usability has to be on a decent level. Administration is done via a website which is beside the Android device the main control system. It is the standard way to connect to the system. That is because nearly every new smartphone is connected to the internet and can visit the website which is not based on the phones operating system. Developers can also make apps for our solution on any Android device which supports ADK (Android Development Kit) with defined interfaces.

Other devices (e. g. IOS Devices, PC clients) are also able to connect to the system using defined web services. It is possible that without knowing the internal structure, these devices can control the whole system.

## 2 Planning

### 2.1 Involved parties

There were four persons involved in this thesis. Split up, there were two teachers and two students.

Dr. Dipl. Ing Thomas Stütz, teacher at the HTBLA-Leonding, was our supervisor on the Java side of the project. He contributed his ideas to the project for the technical implementation, controlled the improvement of the work and he proved, if the milestones were reached in the planned time.

Dipl. Ing Gerald Köck, also teacher at the HTBLA-Leonding, was our assessor at the Arduino side of the diploma thesis. If we had problems implementing some aspects of the system, he led us the right way with his ideas. The general workflow concept is mainly based on his thoughts, but some aspects have been changed.

Meetings with the two assessors were held regularly every week or at least every two weeks.

### 2.2 General approach

We didn't use a certain planning system like the V-Model or complex planning models like that. The approach of ours was to make use of the normal, simple and easy-going project documentation. Documents and information like the project handbook, structure plans or project reports were our guideline for this diploma thesis. During the planning and development we had periodic meetings with our supervisors, for showing them the ongoing advance, discussing further development and for perfecting previous components. Because of that we were able to create and build in new requirements and exceptions, which we had been unaware of before. Therefore our software became more and more adapted to our goals, despite of the growing complexity.

Because of the growing complexity and the overflowing ideas we got, we were forced to prioritize these ideas. By establishing these priorities we implemented and merged them into the system one after another to make sure to get the functionality with the most capital importance to work before getting less significant functionalities done.

### 2.3 Initial situation

#### 2.3.1 Quantity structure of parts

In the point below some components are listed and how much messages they use per day.

- Profiles & Tasks: 50 - 100
- Logging: 150 - 200
- Events: 25 - 50

## 2.4 Goals and requirements

### 2.4.1 Goals of the new system

The following goals should be accomplished by the system:

- Controlling electronical devices in a household
- Automation through associations between sensors and actors
- Remote access over Web-Interface
- Local access over Android-App

This diploma thesis makes it possible to control the electronical devices in a household. This is accomplished by gaining access through a Web-Interface or an Android-based device. To create a certain automation in the system, it is able to make tasks or to define conditions on devices. Through the two mentioned user interfaces it is possible to control these devices in any way. On one of these user interfaces the user is also able to create, edit and delete the above mentioned conditions and tasks.

The web interface will be implemented on a Java EE Server, on which the SOAP - Webservice runs. As a local control station there will be an Android - Tablet.

Because of our interfaces (SOAP, MCP) it is possible to expand the end devices, such as our Java EE Server or Android - Tablet, in any way.

### 2.4.2 Requirements

Requirements of the individual parts of the system:

- Arduino-Microcontroller:
  - Activation piloting of the actors and sensors
  - Persisting of the RCS - Devices on the SD - Card
  - Multiple and simultaneous connections to server/Android-devices
  - Logging of all actions in several modes:
    - \* on SD - Card
    - \* through serial connection
    - \* on the server
    - \* per broadcast to the network
  - Communicating with other devices through our own developed protocol (MCP)
- Java EE Server:
  - Persisting of the RCS - Devices into a database

- Communicating with other devices through our own developed protocol (MCP) or via WebService (SOAP)
- Control and manage the system over a Web - Interface
- Automation through manageable tasks and profiles
- Persist incoming (from Arduino) and own logs into a database
- Android Application:
  - Provide libraries to easily create and connect to our system for other developers
  - Exchanging data with our stationary microcontroller and control the implemented devices

## 2.5 Concept

### 2.5.1 System concept

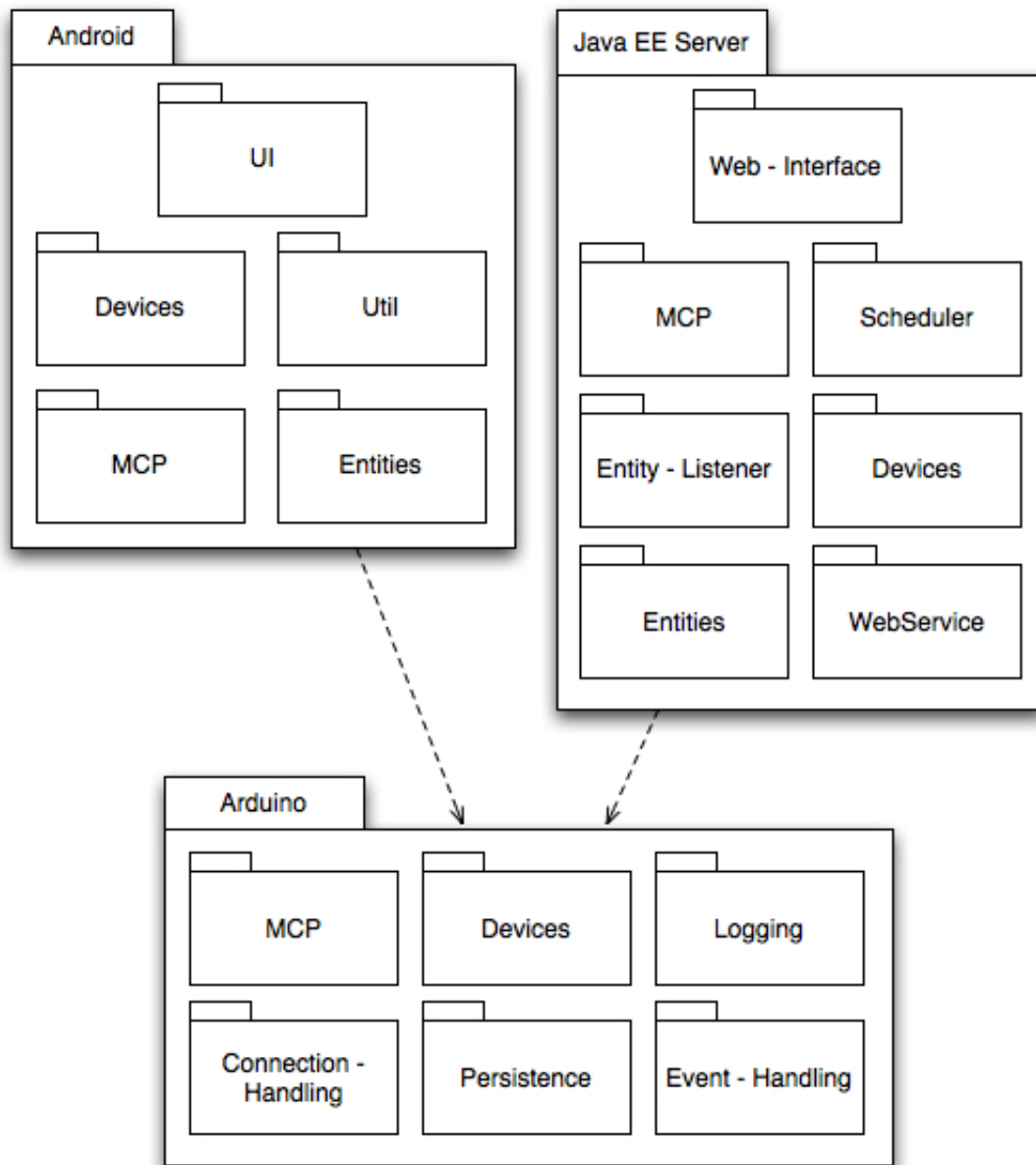


Figure 1: System package diagram



### 2.5.2 Workflow description

There are four workflows which we cover in our solution due to the fact that there is more than one way to connect.

#### Workflow 1

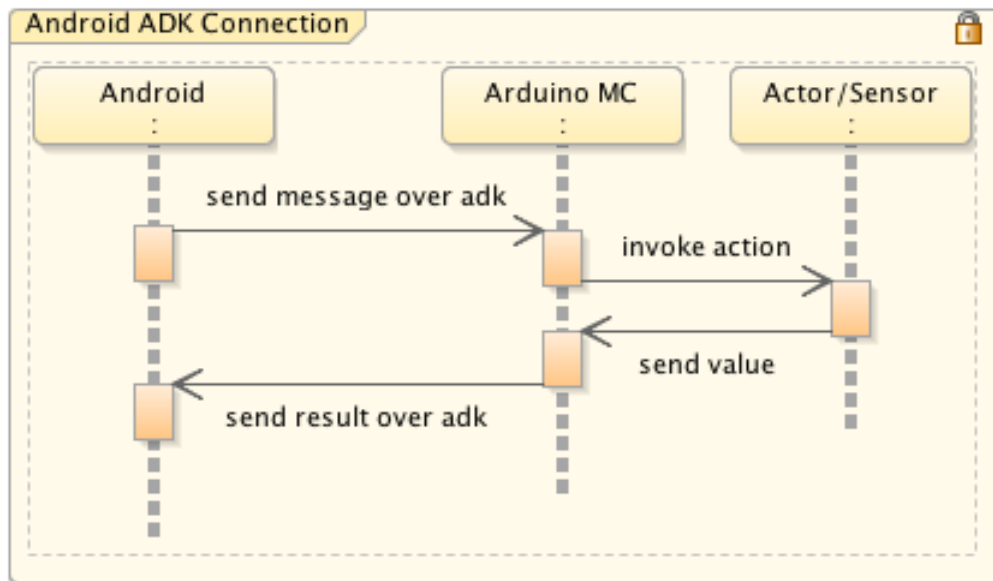


Figure 2: Android ADK Connection

In this workflow the Android unit is connected direct to the Arduino via the ADK framework which allows devices to connect through USB. The Android capable device is used as a stationary monitor and control screen interfacing with an user through a full flexible graphical user interface. User inputs are processed by the app running and then MCP (Minimal Communication Protocol 3.2.2) messages are created and sent through the USB connection. The commands are interpreted by the Arduino microcontroller and then the actions are applied to the corresponding actors or sensors. Example applications for this workflow would be a light control system or a rollerblind management system or a system where both are combined.

## Workflow 2

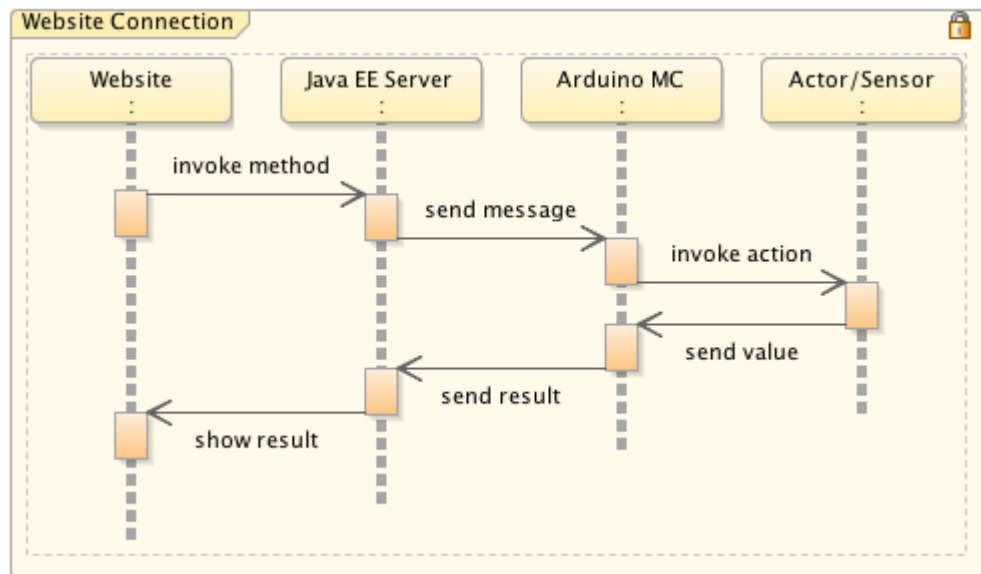


Figure 3: Website connection

This workflow describes the generic access to the system with the web interface. It is possible to connect from everywhere with a client capable of browsing the internet. The user interface offers the advantage to manage all devices, check their states and read the global Sensor values. The webserver then processes the input, generates MCP messages and sends them to the defined IP-Address. IP-Addresses of servers are administrated within the website. The Arduino microcontroller then interprets the commands and the actions are then applied to the corresponding actors or sensors. In addition all the administrative task can be done in the Java EE application and a basic security authentication is also provided.

## Workflow 3

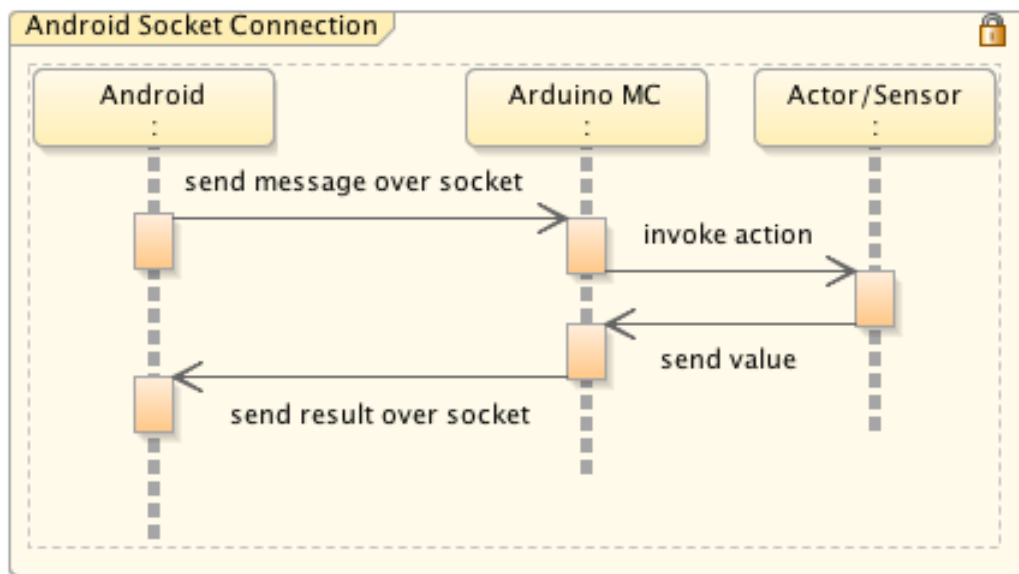


Abbildung 4: Android Socket Connection

If the Android unit is not direct connected to the arduino base station, our implementation offers the availability to connect through a socket connection. We provide our own Java source files which allow easy connection to it without knowing the structure of MCP. The connection is established direct to the Android unit and commands are sent to it. In the best case, the Android app detects when a direct connection is available and changes the connection type.

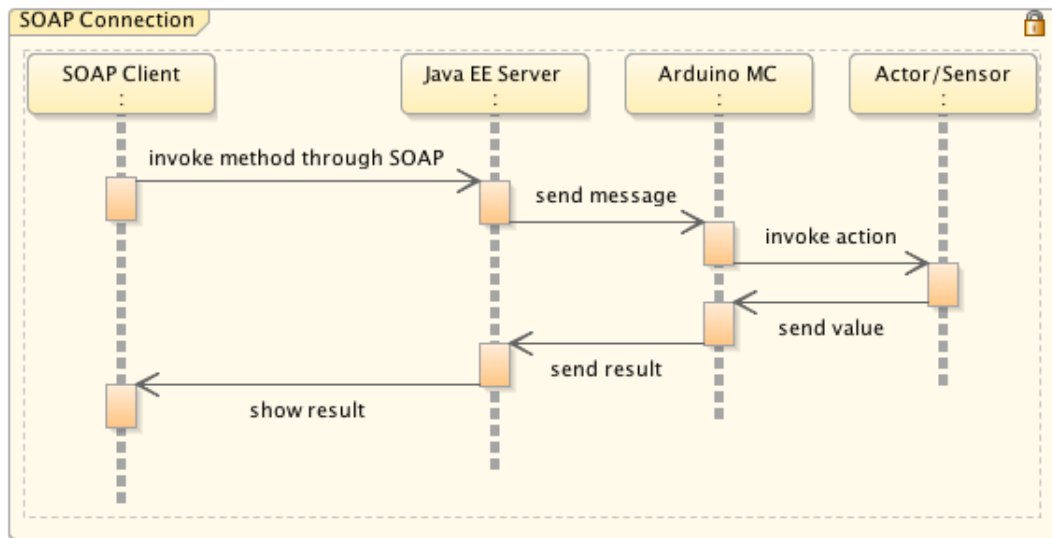
**Workflow 4**

Figure 5: SOAP Connection

Any other device, for example IO based devices or Windows phones can connect to the control unit via defined SOAP interfaces. Through these interfaces we offer the ability to use the exact same functions as used when connection is direct via USB. This workflow was created because otherwise, any device which is not Android based must implement the whole protocol to connect to it. Instead, the device connects via SOAP to the main station and the Java EE server interprets the request and sends it to the Arduino microcontroller.

**2.5.3 System architecture**

To exploit the benefits from the backend to the frontend the best, the system architecture is built heterogeneously. Figure 6 shows an overview.

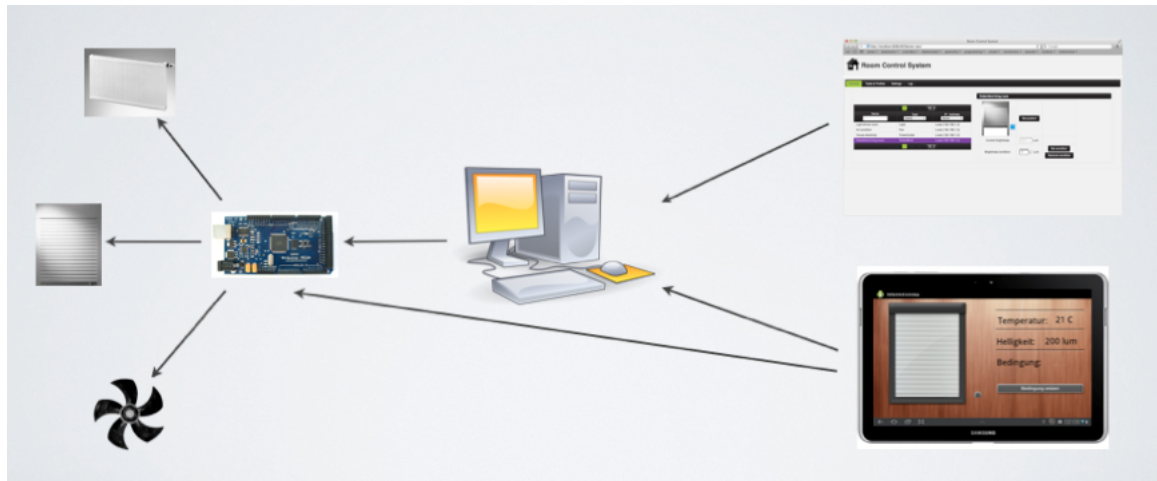


Figure 6: System architecture

### Arduino

Processor	ATmega2560
SRAM	8 KB
Operating Voltage	5V
Clock Speed	16 MHZ

Table 1: Power data Arduino

### Java EE Server

Processor	2,4 GHz Intel Core 2 Duo
Memory	4 GB 1067 MHz DDR3
Hard disk	250 GB
Operating system	Mac OS X Lion 10.7.3

Table 2: Power data Java EE Server

### Database

The included „Apache Derby“ is used as database with the integrated derby network client.

The Derby network client provides network connectivity to the Derby Network Server. It is distributed as an additional jar file, derbyclient.jar, with an entirely independent code base from the embedded driver.

### **Application Server**

As our application server we use „GlassFish v3.1.2“.

GlassFish is a reference implementation for Java EE and as such supports Enterprise JavaBeans, JPA, JavaServer Faces, JMS, RMI, JavaServer Pages, servlets, etc. This allows developers to create enterprise applications that are portable and scalable, and that integrate with legacy technologies. Optional components can also be installed for additional services.

### **Web service**

SOAP, originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It relies on Extensible Markup Language (XML) for its message format, and usually relies on other Application Layer protocols, most notably Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

### **Clients**

As a client any operating system can be used and also any browser can be used.

## 2.6 Project planning

### 2.6.1 Project structure

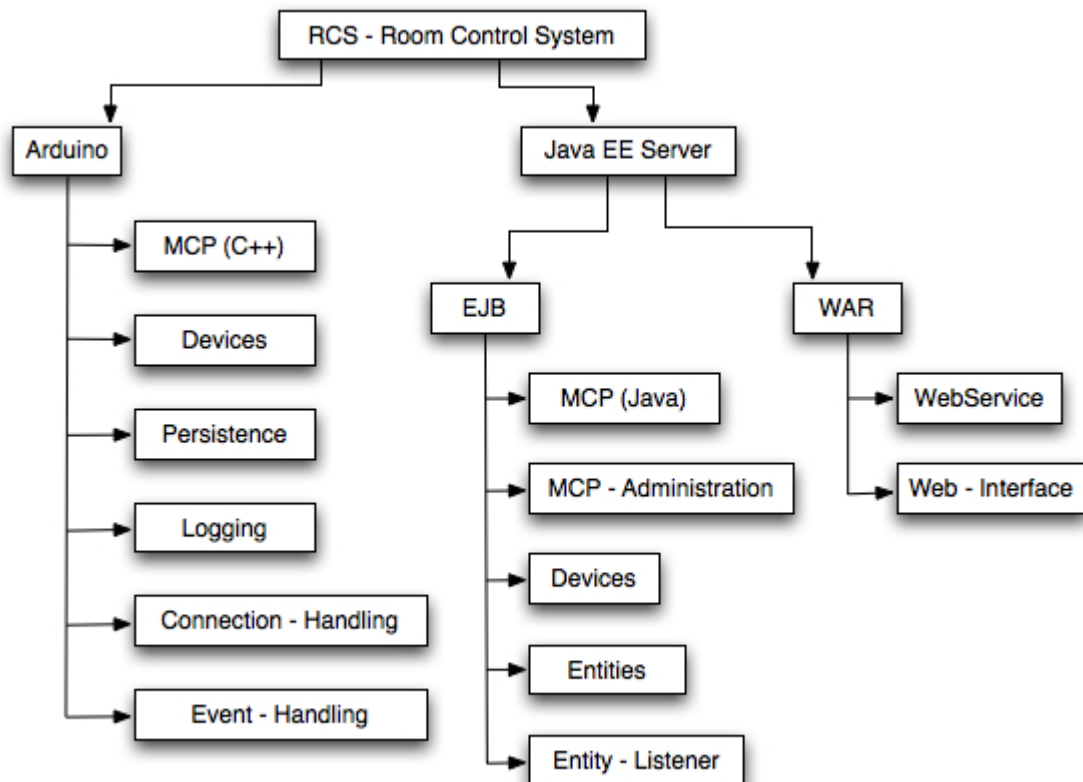


Figure 7: Project structure plan

In Figure 7 the project structure is displayed. During the implementation it was essential to develop the modules simultaneously, because they had to communicate among one another and therefore they are closely linked. This kind of project structure plan is called „mixed project structure plan“, which means that it is first divided by modules and after that by work packages and corresponding development phases.

### 2.6.2 Time scheduling

#### Milestones

Based on the project structure plan the work effort of the work packages was estimated. Following the appointments for the milestones could be determined. Table 3 shows the

work packages and the planned completion.

Work package	Planned completion
Collecting ideas -> concept	Beginning October
Project handbook	Mid October
Creating and implementing data model	End October
Implementation of connection - and event - handling	End November
Implementation entities and entity - listener	End November
Debriefing	End November
Implementation of MCP	Mid January
Implementation of persistence and logging	Mid January
Debriefing	Mid January
Implementation of web service and parts of web interface	Beginning March
Implementation of devices	Beginning March
Debriefing	Beginning March
Complete implementation	Mid March
Tests and changes	End March
Completion	Beginning April

Table 3: Milestones



## 3 Implementation

### 3.1 Description of used technologies

#### 3.1.1 Java 6

*„Java is a programming language expressly designed for use in the distributed environment of the Internet. It was designed to have the "look and feel" of the C++ language, but it is simpler to use than C++ and enforces an object-oriented programming model.“ [4]*

#### Bytecode and machinecode

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any hardware/operating-system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead direct to platform-specific machine code. Java bytecode instructions are analogous to machine code, but are intended to be interpreted by a virtual machine written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a web browser for Java applets.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would. Just-in-Time compilers were introduced from an early stage that compile bytecodes to machine code during runtime.

#### 3.1.2 JPA

The Java Persistence API (JPA) is an interface for Java applications, which simplifies the assignment and transfer of objects to database records. It simplifies object-relational mapping, which is about saving objects in a Java application into a relational database which are not originally designed for object-oriented data structures. The Java Persistence API was developed as a project of the JSR 220 Expert Group and first released in May 2006. Hibernate had great success and broad acceptance in the developer community. So the JSR220 built on the principles of Hibernate. [7]

#### 3.1.3 Oracle GlassFish Application Server v3.1.2

GlassFish was originally an open-source project that was originally developed by Sun Microsystems. In addition to the open source version a commercial version was developed. The differences between these two versions are irrelevant to this diploma thesis. The commercial version includes enhancements such as clustering and administration tools.

The Application Server provides a runtime environment for Java applications and web services. Not only the Java Standard Edition (JavaSE) is used, but also the Java Enterprise Edition (Java EE).

Among other things GlassFish Application Server supports following modules:

- Enterprise Module: An enterprise module or enterprise application called, is part of Java EE and refers to a collection of servlets, EJBs, HTML pages, classes and other resources that can be uploaded to an application server, so that they can work together. The difference to a web module is that the EJBs and HTML pages are split up in two different projects. These components will not be uploaded individually, JAR and WAR file, then these two are packaged into an EAR file and then uploaded.
- Web Module: A web module or web application called, is part of Java EE and refers to a collection of servlets, EJBs, HTML pages, classes and other resources that can be uploaded to an application server, so that they can work together. These components will not be uploaded individually, but packaged into a WAR file and then uploaded.
- EJB Module: An EJB module is an executable software unit, which consists of one or more enterprise beans. The standard format for this module is the JAR archive.

In the present work an enterprise module was used, which accesses the database via JPA.

#### 3.1.4 Apache Derby

Apache Derby is a project of the Apache Software Foundation, which developed a Java-based relational database management system. Apache Derby is one a lightweight database, as it has the delivery of only two megabytes in size and is easily installable. Derby is mainly, but not exclusively, used in Java projects.

The software was originally developed by the company Cloudscape Inc., in Oakland (California) under the name JBMS. The first version was released in 1997. The product was later renamed to "Cloudscape". 1999 Cloudscape was bought by the company Informix Software Inc., and in 2001 the database division was acquired by IBM.

In 2004 IBM transferred Cloudscape to the Apache Software Foundation software under the name "Derby" as free software. As early as 2005 Sun Microsystems participated in Derby. In 2006 Derby was integrated in the Java Development Kit as Java DB from Java 6 and is now also supported by Sun Microsystems.

#### 3.1.5 Arduino

Arduino is a descendant of the Open Source Wiring Platform designed to make electronic components more accessible. Wiring is a development platform consisting of three parts: The programming language, the integrated development environment and a single board

microcontroller. The language basically consists of C/C++ although there is no `main()` function and a few minor differences. Instead developers have to define a `loop()` and a `setup()` function. These files are called sketches and have the endings `.pde`. Part of the language is the wiring library which offers common input/output functions.

Every sketch has the same structure:

---

**Algorithm 1** Arduino sketch file

---

```
#define
#include <Libraries>
void setup()
{
    //initialise routines
}
void loop()
{
    //method is called in an endless iteration
}
```

---

The IDE is cross platform software written in Java and so available for every operating system where Java is available. The hardware itself is entirely open-source meaning that the whole circuit layout can be modified and customized. The framework on which Arduino is based had been ported to various microcontrollers for example the ARM Cortex R3. Without any changes on the software side, it is possible to change the controller underneath. Specific functions only available for the current controller must not be used, otherwise you cannot change it. As mentioned before, developing for the Arduino allows the developer to use on the one side platform unspecific using functions from the framework, on the other side it can be very controller specific when functions are not available otherwise.

As of May 2011, more than 300,000 Arduino units are "in the wild." [10]

This shows that the Arduino is becoming more and more popular. The main reason for this is that even for people which are not used to electronics, it offers the ability to interface with hardware in a relatively easy way. Official Arduino candidates are based on the megaAVR series of chips specifically the ATmega8, ATmega168, ATmega328, ATmega1280, and ATmega2560. In the standard software there is a bootloader included which simplifies the developing process and there's no need for an external programmer. (Appendix A: Arduino Models)

So called shields are available for the Arduino which are exactly fitting onto the open ports, it is stackable. This means that e. g. an ethernet shield which is used in our solution can be easily plugged in and programmed. The platform has a huge user base with tons of libraries available for free in the internet.

„Arduino was built around the Wiring project of Hernando Barragan. Wiring was Hernando’s thesis project at the Interaction Design Institute Ivrea. It was intended to be an electronics version of Processing that used our programming environment and was patterned after the Processing syntax. It was supervised by myself and Massimo Banzi, an Arduino founder. I don’t think Arduino would exist without Wiring and I don’t think Wiring would exist without Processing. “ [9]

The Arduino syntax is patterned after the Processing syntax. Processing is an open source programming language and an integrated development environment built for graphic, simulation and animation programs. It is a simplified version of Java allowing easy interactions with graphical elements. Essentially, the programming language introduced in the Arduino IDE is a simplified version of C/C++.

### **3.1.6 Android SDK**

Android is an open source operating system developed mainly for smartphones and in a row also for tablets with version 3.0 and up.

#### **Smartphones**

A smartphone is a mobile phone offering many more capabilities such as surfing the internet or chatting. Basically said, it offers capabilities getting close to a personal computer.

The functions can be often extended by so called apps. To make that possible, smartphones have open API offering functions to bring the capabilities to a maximum. Any interested developer can download this API and write his own apps for the smartphone based OS. The apps are then OS specific, meaning that opening an IOS application in Android would cause an exception. There were attempts to make a platform independent API but the specific functions can then not be used. The developed apps were then distributed via a market system. In Android it is the Play Market and the IOS equivalent is the AppStore. Interested users can then browse the offered applications by categories and download new apps.

#### **Tablets**

A tablet is basically a portable computer without a keyboard. User inputs are made via a stylus or via a touch sensitive screen. This defines nothing about the operating system running on it. The first tablets which came on the market had a Desktop OS supplemented by a number of drivers and softwares supporting the touchscreen. Over time, tablets got their own operating system derivatives customized and optimized for tablets. Of course, there are a few tablets nowadays based on Desktop OS’ but these are

the minority. The two main counterparts on the tablet sectors are Android and IOs. The principle of distributing apps remains the same, but apps have to be specially customized to fit into the tablet user's needs.

The programming language for developing apps using the specific platform API is a dialect of java interpreted from an Dalvik named Interpreter. It was originally written by Dan Bornstein and improved over time. The apps are compiled to Java bytecode and then converted in the Dalvik .dex format. This format is designed to fit in environments with low storage capabilities and limited resources.

### System architecture

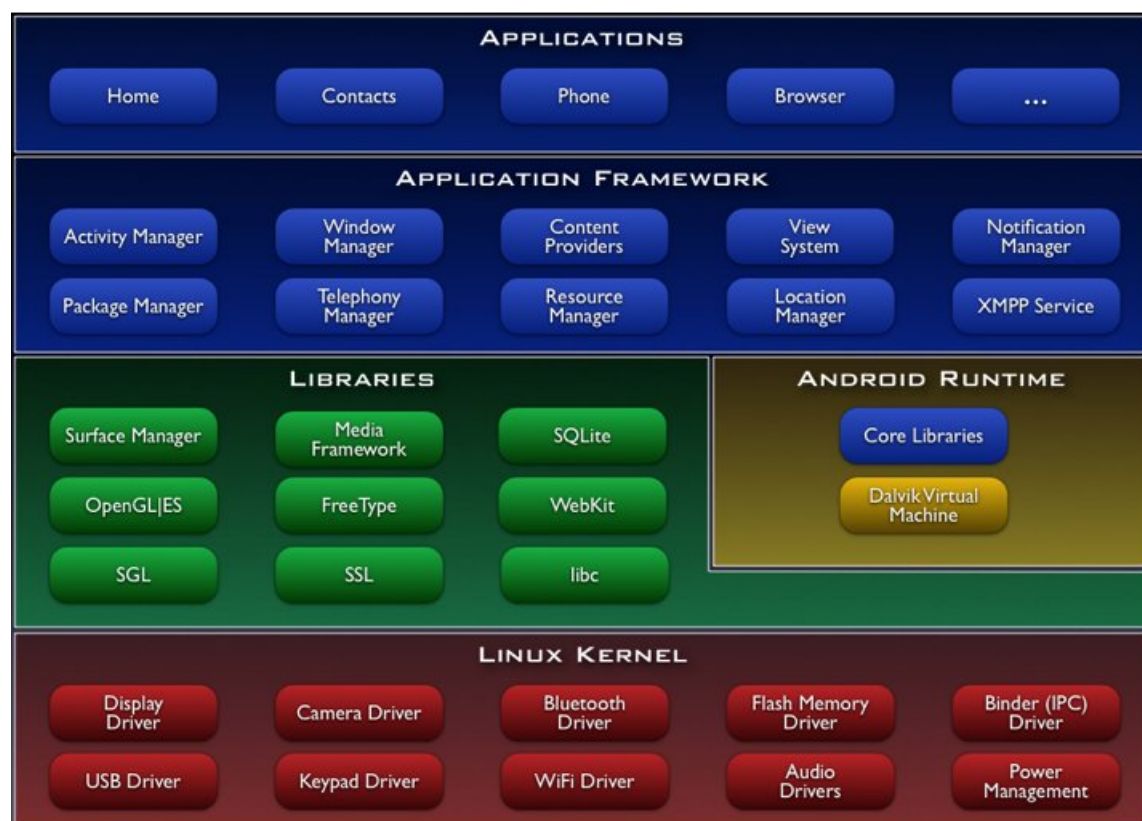


Figure 8: Android System Architecture

The **topmost layer**, named „Application“ is where apps can be run using the functions of the Android API. There are different APIs to choose from but any level is backward compatible. Multiple apps were run simultaneously at the same time.

The **Application Framework** is used for managing the applications, switching between them and offering system wide services such as GPS.

The **Libraries** are called if an app needs them. Examples for such libraries are OpenGL which is used for creating 2/3D view or SQLite which allows apps to have their own small database locally on the device.

The **Android Runtime** interprets the Dalvik .dex files and compiles them into native code. It links the libraries and handles tasks like memory management and exception management.

**Differences between native and Java bytecode** Java bytecode is platform independent, this means that when a Dalvik interpreter is available on the system, the Java bytecode can be run. So it is possible to write a Dalvik interpreter for Windows or Linux. On compile time, or before the application starts, the Java bytecode will be compiled into native code which then can be run on the processor. Normally, in native code exists no garbage collector and the programmer has to detect memory leaks to ensure the stability of the program. In Java bytecode, this is done automatically by the interpreter.

The last layer is the linux kernel layer. Due to the fact that Android is based on a Linux kernel, system specific drivers have to be developed for the system functions such as the Audio driver or the power management.

### 3.1.7 ADK (Android Open Accessory Development Kit)

The Android 3.1 platform (also backported to Android 2.3.4) introduces Android Open Accessory support, which allows external USB hardware (an Android USB accessory) to interact with an Android-powered device in a special "accessory" mode.[6]

Basically, it allows Android devices to connect to a peripheral device and it offers the ability to communicate with this device. In our case, this device is a special Arduino version designed to use with an ADK compatible smartphone. Once connected, it allows a two way communication between Android and the Arduino. Due to the fact that not every Android smartphone or tablet is able to establish this connection, a number of routines have to be run through to determine it.

The following function has to be called to get the permission to connect to the device:

---

**Algorithm 2** Arduino starting permission intent

---

```
UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB_SERVICE)
    ;
//retaining the UsbManager SystemService
private static final String ACTION_USB_PERMISSION = "at.roomControllSystem.
    USB_PERMISSION";
//defining the appropriate permission string as a constant

//insert logic

mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(
    ACTION_USB_PERMISSION), 0);
//retaining the intent for the equivalent action
IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);
//filtering only actions concerning this object
registerReceiver(mUsbReceiver, filter);
//registering the created reciever Object
mUsbManager.requestPermission(device, mPermissionIntent);
//starting the request
```

---

An intent is an abstract description of an operation to be performed. The intent which was used opens a dialogue asking for the permission to use the usb device. To get intents which were addressed to this class, a filter was created defining that it only receives USB permissions. The last step is requesting the permission using the global UsbManager service. To receive to intent, the following BroadcastReceiver has to be created:

---

**Algorithm 3** Arduino BroadcastReceiver

---

```
private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver()
{
    //This code will be executed when the Broadcast is received
    public void onReceive(Context context, Intent intent)
    {
        String action = intent.getAction();
        //checks whether the broadcast belongs to this application
        if (ACTION_USB_PERMISSION.equals(action))
        {
            //used for synchronizing threads
            synchronized (this)
            {
                //retains the UsbDevice
                UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.
                    EXTRA_DEVICE);
                if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED,
                    false)) //checks if the permission is granted
                {
                    if(device != null)
                    {
                        //sets up the connection
                        setupConnection();
                    }
                }
                else
                {
                    //Log to the Android logging output
                    Log.d(TAG, "permission_denied_for_device_" + device);
                }
            }
        }
    }
};
```

---

It reacts on the result of the query and detects if the permission is granted or not. It logs the result and calls `setupConnection` to establish the connection. `setupConnection` connects and send one bulk data packet to start the transfer.



---

**Algorithm 4** Arduino Bulk Transfer

---

```
//retrieves the interface  
UsbInterface intf = device.getInterface(0);  
//gets the Endpoint  
UsbEndpoint endpoint = intf.getEndpoint(0);  
//Opens the device and retains a connection object  
UsbDeviceConnection connection = mUsbManager.openDevice(device);  
//claims the interface and ensures that it's used only by us  
connection.claimInterface(intf, forceClaim);  
//sends a bulk message to ensure the Arduino is ready  
connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT);
```

---

## 3.2 Technical Solution

### 3.2.1 System Architecture

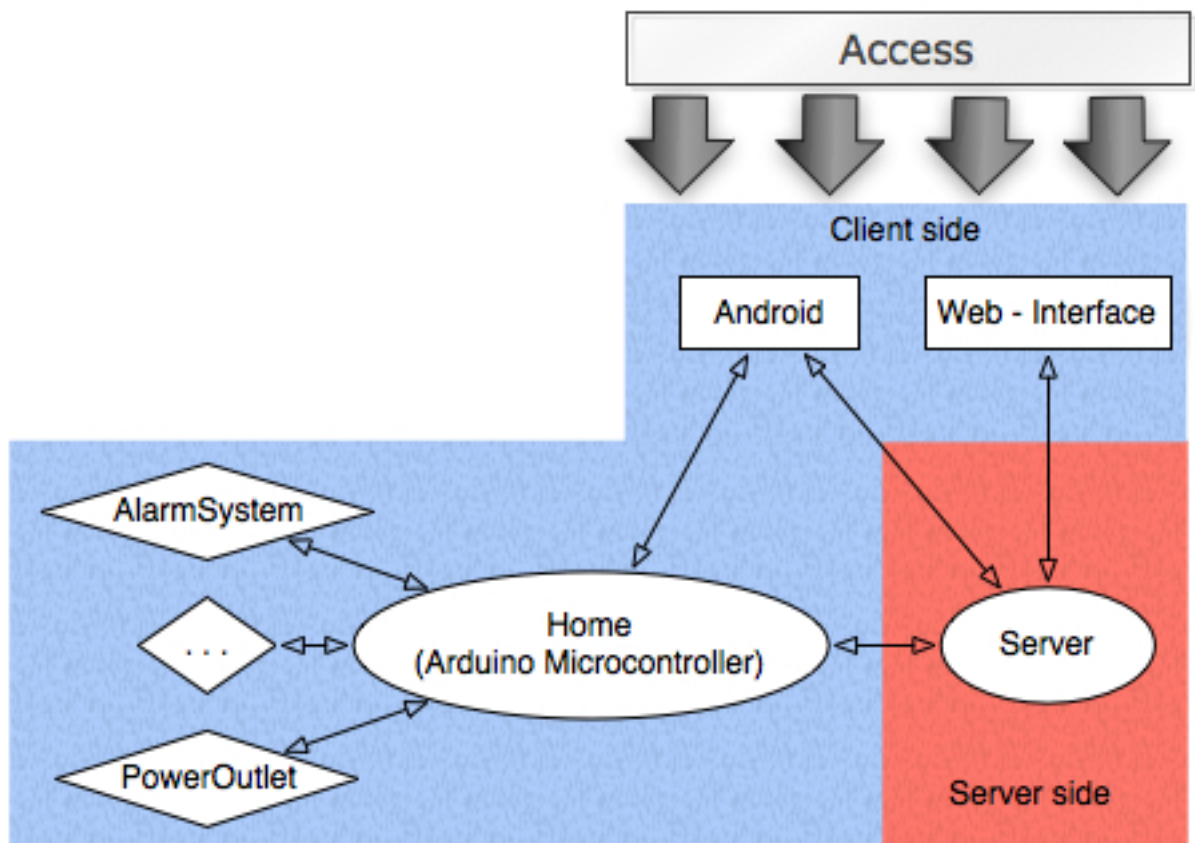


Figure 9: System architecture

### 3.2.2 MCP (Minimal Communication Protocol)

The “Minimal Communication Protocol” (MCP), is responsible for the communication between the Java Server / Android device and the Arduino MC. It consists of the following message types:

- CallMessage = call a method
- LogMessage = to log something
- DeviceMessage = represents a device
- ResultMessage = returns one or more results from a call

The Message ID is a system-wide identification number, which is used for a unique identification of a message.

#### CallMessage

A CallMessage is used for calling a method on the Arduino side and also on the Java EE side. Methods can have parameters and method overloading is also possible. A CallMessage has a simple general structure:

#<MessageId>:<MessageType>:<DeviceId>:<MethodName>:<parameterlist>;

**MessageId** is a system wide number and is generated continuously using the GlobalMessageID class. Every following message has a higher number than the one before. If they are not synchronized, the next time when one of the client sends a message, they will synchronize again.

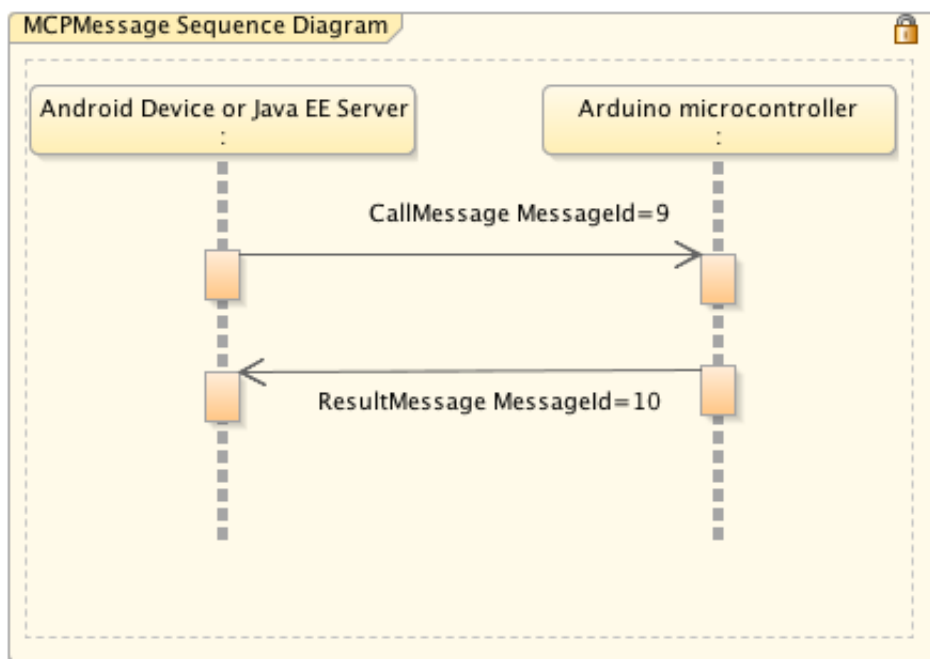


Figure 10: MCPMessage Sequence Diagram

**MessageType** is an enum describing the type of the message. MessageTypes can be CallMessage, LogMessage, DeviceMessage or ResultMessage.

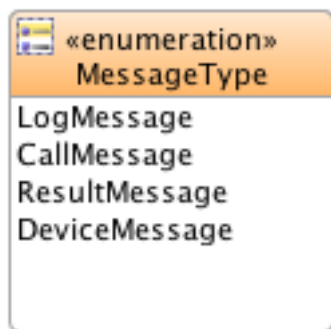


Figure 11: MessageType Enum

**deviceId**: methods called are specific to the device because not every device has the same methods. The methods are invoked dynamically at runtime. The client can call

getDevices to get all available devices with their current status. Therefore the DeviceId must be set to 0, calling global methods.

Example call to get all devices:

```
#25:1:0:getDevices();
```

**MethodName** describes the method which the user wants to call. It has to be a device specific method. If the method cannot be invoked, the result will be for example #26:2:3:Method not found;

There can be as many parameters as the function needs allowing function overloads in this protocol. This means that calling a function with 2 parameters will result in a different answer to calling the same function with only one.

Here are some examples for CallMessages:

```
#30:1:10:turnOn(); //In this case the device Called is a PowerOutlet.
```

```
#38:1:11:setPosition(200); //sets the position of a rollerblind
```

## LogMessage

Every Home Automation System has its own logging mechanisms because they are running 24 hour a day, 7 days a week. Aside basic things like a watchdog protection, logging is also implemented using the following structure:

```
#<MessageId>:<MessageType>:<DeviceId>:<LogLevel>:<LogMessage>;
```

**MessageId** is a system wide auto generated number described above.

**MessageType** is in this case 0 for LogMessage.

**DeviceId** is the device which autonomously sends the log message or the global device 0.

**LogLevel** is an enum describing the level of the event. The enum has the following members:

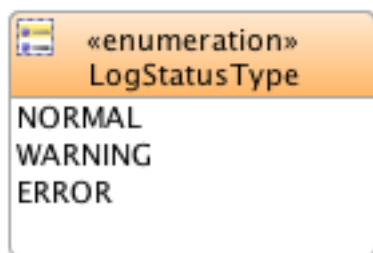


Figure 12: LogStatusType

Here are some examples for LogMessages:

```
#30:0:10:0:SSR turned On;
```

```
#38:0:0:2:System reset;
```

### DeviceMessage

DeviceMessages are only sent as answers to an incoming CallMessage calling getDevices(). All available devices are then sent continuously until every device's representation is sent. The structure of a DeviceMessage:

```
#<MessageId>:<MessageType>:<DeviceId>:<DeviceType>:<DeviceName>:<CurrentState>;
```

**MessageId** is a systemwide auto generated number described above.

**MessageType** is in this case 3 for DeviceMessage.

**DeviceId** describes the Id of the device which is going to be transmitted.

**DeviceType** is the type of the device.

**DeviceName** is a String limited to 40 characters describing the device for example „Light Livingroom“ or „all Rollerblinds upstairs“.

**CurrentState** is a String consisting of a number of properties formatted as property-name=propertyvalue;propertyName=propertyvalue;..... There can be as many properties as needed.

An Example DeviceMessage would be:

```
#2:3:1:Light Livingroom:turnedOn=false;
```

### ResultMessage

After a CallMessage is sent, a ResultMessage is sent to ensure the new state of the device. ResultMessages are only sent as results to CallMessages not on once own.

```
#<MessageId>:<MessageType>:<DeviceId>:<Result>;
```

**MessageId** is system wide auto generated number described above.

**MessageType** is in this case 2 for DeviceMessage.

**DeviceId** describes the Id of the device where the Method is called, or 0 for the global system Device.

**Result** is a String describing the new state of the Device.

An Example DeviceMessage would be:

```
#201:2:5:Light has been switched off;
```

```
#202:2:5:OK;
```

### 3.2.3 Data model Java EE Server

#### MCP - Activity data model

Figure 13 shows an overview over the MCP - Activity relation.

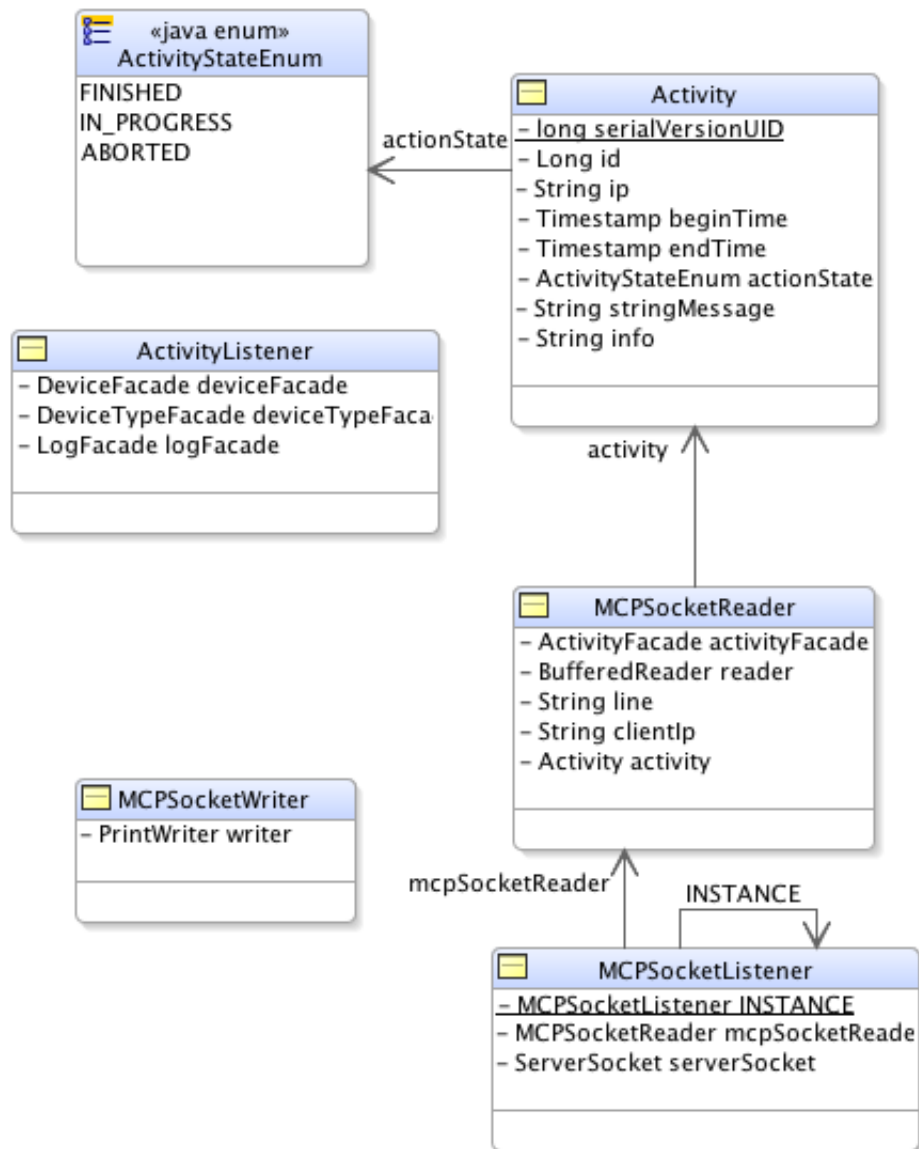


Figure 13: MCP-Activity data model

## MCP Message data model

Figure 14 shows an overview over the MCP Message data model.

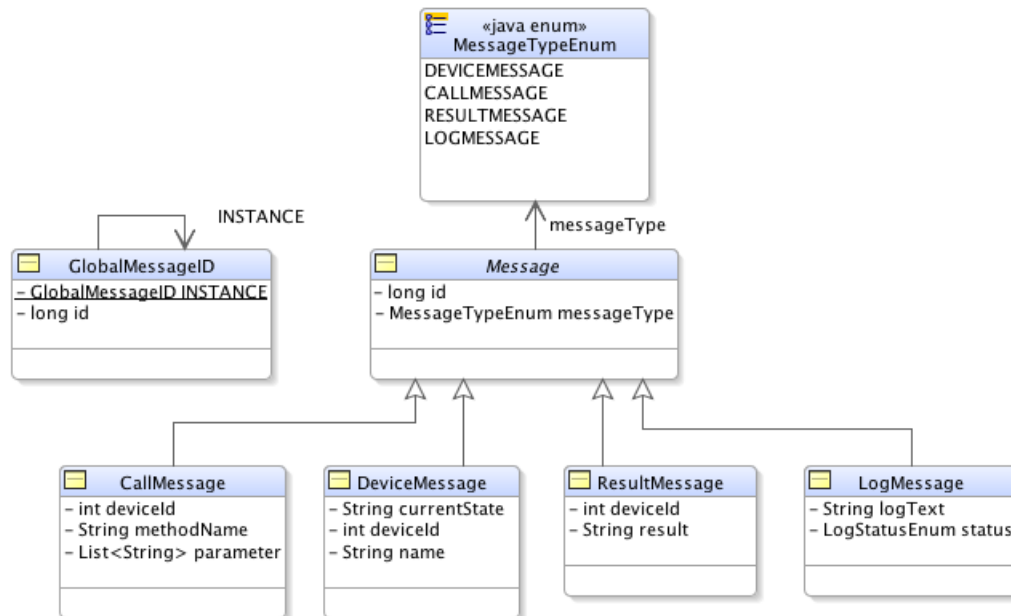


Figure 14: Message data model

## Devices data model

Figure 15 shows an overview over the device data model.

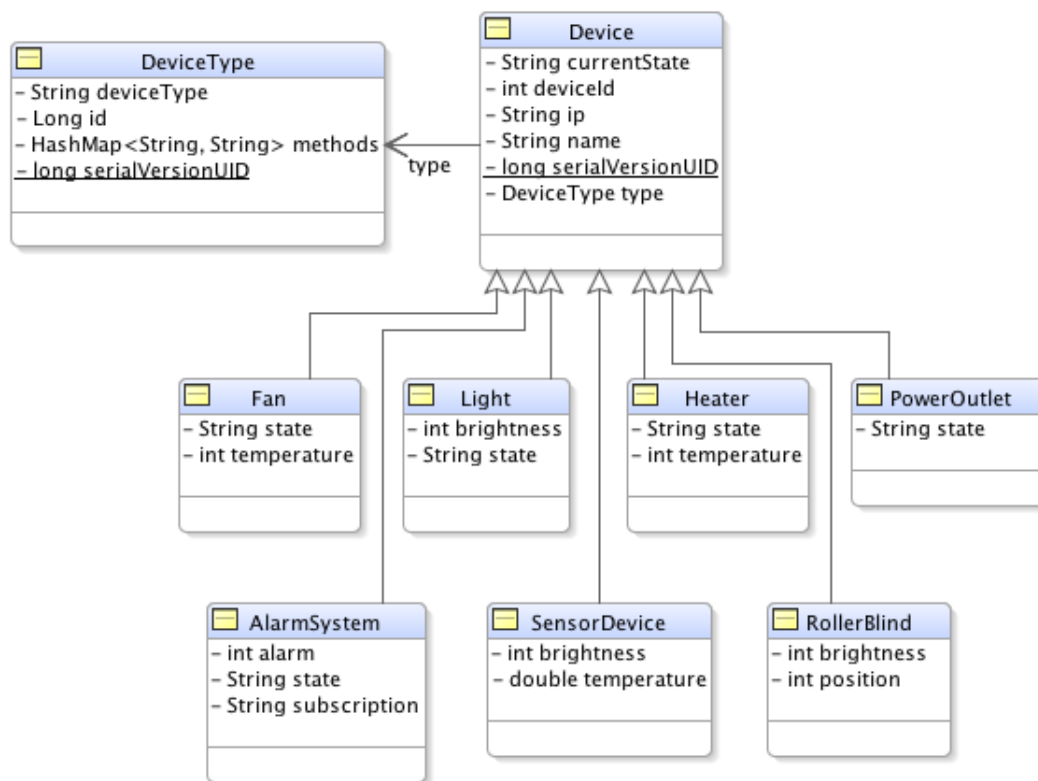


Figure 15: Device data model

### Table data

- ACTIVITY
- DEVICE
- DEVICETYPE
- TASK
- PROFILE
- PROFILE\_TASK
- LOG

### Table ACTIVITY:

This is one of the central tables on the server, because every incoming message will be recorded in an activity. That is why it is indirectly linked to the **DEVICE** and **LOG** tables, because these entries will automatically be created by a successfully processed activity.



- **BEGINTIME**

This property is a date with time and it will be set when the activity is created, therefore when the message is received.

- **ENDTIME**

This property is also a date with time and it is set when the processing is done.

- **ACTIVITYSTATUS**

This is an enum and it is set when the processing is done.

It has the following three types:

- **FINISHED**: Is used when the activity is executed without any problems.
- **IN\_PROGRESS**: Is used when the activity is still in execution.
- **ABORTED**: Is used when the activity had an error during the execution.

- **IP**

This is a string and it contains the IP-Address from where the message came.

- **STRINGMESSAGE**

This is a string and it contains the received message.

- **INFO**

This is a string and contains information what was done and accordingly what happened.

#### **Table DEVICE:**

The devices are the representation of the real electronical devices in the household.

- **DEVICEID**

This is a unique integer which represents a device.

- **NAME**

This is a string and the name of the device.

- **CURRENTSTATE**

This represents the current state of the device in a string. Such as if it's on or off.

- **IP**

That is the IP-Address where the device is located. Persisted as a string.

- **TYPE\_ID**

This is a foreign key to the DEVICETYPE table.

#### Table DEVICETYPE:

In this table there are the different types of devices.

- DEVICETYPE  
Represents the type of a device as a string.

#### Table TASK:

Represents the tasks which the user creates.

- DAYS  
This is a boolean array of seven elements. It represents the days on which it should process the task.
- TASKTIME  
It is the time in which the task is processed.
- TODO  
Regulates the actions to do. It is persisted as a list of CallMessages.

#### Table PROFILE:

Profiles are a bundle of tasks, for making it easy to switch different behaviours of the system.

This table contains:

- NAME  
This is the name of the profile.
- ACTIVE  
Is a flag for the current active profile.
- TASKLIST  
Is the list of tasks in the profile.

#### Table PROFILE\_TASK:

Is the associative table between task and profile, because they have a unidirectional many to many relationship.

### Table MCPPlace

Each Arduino MC is a own MCPPlace. A MCPPlace has a place, an IP - address and a boolean if logging is activated

- PLACE  
Represents the name of this object as a string.
- IP  
The IP - address of the Arduino MC, persisted as string.
- LOGGINGACTIVATED  
A boolean if the server should receive log messages from this Arduino.

### Table LOG:

- LOGSTATUS  
This is an enum and it has the following three types:
  - NORMAL: A normal log with no exceptional behaviour.
  - WARNING: Is used when something unexpected happens.
  - ERROR: Is set when critical things happen.
- LOGTEXT  
The text of the log.

## 3.3 Description of the system components

### 3.3.1 Used Libraries on Arduino

#### Standard template library

The standard C++ library is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed. [2]

The Standard Template Library is part of the standart C++ library. Normally, as in every newer operating system, the Standard Template Library, or STL is ported, available and tested by many different persons. Ports for the Atmega systems are instead only rarely available and there are only a few wich are nearly bugfree. The version which we included is from <http://andybrown.me.uk/ws/2011/01/15/the-standard-template-library-stl-for-avr-with-c-streams/>

## Why are we using STL?

STL provides a number of functions and templates which are useful for our system. If we wouldn't use STL, we would have to port much things by ourselves rather than using this library. We used Vectors, Queues, Maps, lists, comparers and iterators.

## Vectors

Vectors are basically arrays which can handle as much elements as free memory is available. They achieve this through reallocating and allocating ahead strategies for faster adding of elements. Vectors are defined as follows:

---

### Algorithm 5 Arduino Vector Definition

---

```
std::vector<int> vec;
```

---

As seen, the vector class, as well as the map and the list class uses templates for generating type based classes. Instead of, like in .NET generating only one class and then dynamically associating them, every new template has to be generated and stored on the microcontroller which means that the flash memory used will increase rapidly when dealing with different types of arguments.

Elements can then be added to the vector with one simple line of code:

---

### Algorithm 6 Arduino Vector Add Element

---

```
vec.push_back(3);
```

---

The order of the elements will still remain the same, due the fact that the element is added at the back.

In .NET, iterating through elements is done with a foreach loop. In C++, it is a little bit different:

---

### Algorithm 7 Arduino Vector Iterator

---

```
std::vector<int>::iterator it;  
for (it=vec.begin(); it!=vec.end(); it++)  
{  
}
```

---

Iterators are used for iterating through the elements by only calling it++.

## Lists

Lists are nearly handled as vectors with the exception that the elements are stored otherwise. List elements are stored with references to the next element which gives it the ability to be faster when iterating through them. Lists elements are added the same way and they are deleted the same way. They are defined using:

---

**Algorithm 8** Arduino List Definition

---

```
std::list<int> list;
```

---

Iterating through them is also nearly the same as iterating through vectors with the exception that the type of the iterator has changed to

---

**Algorithm 9** Arduino List Iterator

---

```
std::list<int>::iterator it;
```

---

## Maps

Maps are a kind of associative container that stores elements formed by the combination of a key value and a mapped value. [3]

In .NET, maps are known as dictionaries. Data in maps, or dictionaries are always defined as key-value pair. Maps can be defined as described beneath:

---

**Algorithm 10** Arduino Map Definition

---

```
std::map<char*,char*> map;
```

---

In this variant, the map uses the standard comparer function, comparing only by references. If the key is, like in our case a char array, comparing them with references may result in the wrong value for the key. The solution is provided within the STL library.

---

**Algorithm 11** Arduino Map Comparer

---

```
//defines a custom comparer struct
struct str_cmp {
    bool operator()(char const *a, char const *b)
    {
        return strcmp(a, b) < 0;
    }
};
//defines the map with a custom compare function
std::map<char*,char*,str_cmp> map;
```

---

Maps values be accessed and inserted using the following syntax:

---

**Algorithm 12** Arduino Map Element Access

---

```
map["key"] = "value"; //if key and value are of type char*  
//or  
map[2] = 32.12; //if key is of type integer and value is of type double
```

---

Iterating through the container is as simple as iterating through lists.

---

**Algorithm 13** Arduino Map Iterator

---

```
std::map<int, char*>::iterator itx; for ( itx=deviceMap.begin(); itx !=  
    deviceMap.end(); itx++ )  
{  
    int key = itx->first;  
    char* value = itx->second;  
}
```

---

## Queues

A queue is based on the First In First Out (FIFO) prinzip, meaning that elements that were enqueued were added to a kind of waiting queue. Only the first element can be accessed. Defining queues is pretty straight-forward.

---

**Algorithm 14** Arduino Queue Definition

---

```
std::queue<int> queue;
```

---

Elements are enqueued using this line of code

---

**Algorithm 15** Arduino Queue Add Element

---

```
queue.push(3);
```

---

Once inserted, the first element can be accessed with the line

---

**Algorithm 16** Arduino Queue Dequeue Element

---

```
int a = queue.pop();
```

---

Note that once this line is finished, it dequeues this element from the list.

## SdFat

SdFat is an Arduino library that supports FAT16 and FAT32 file systems on standard and high capacity SD cards. SdFat supports file creation, deletion, read, write, and truncation. SdFat supports access to subdirectories, creation, and deletion of subdirectories. [1]

The SdFat library is used to save the devices in a readable and approved format. It is supported by any newer operating system and therefore it's perfectly fitting or needs. This library can be found at <http://code.google.com/p/sdfatlib/>. It is commonly used in Arduino projects and therefore is approved by many other programmers.

## Comonly used operations

### Opening a file

---

#### Algorithm 17 Arduino Opening SdFat File

---

```
//note that SPI_HALF_SPEED is used because there are less write exceptions
if (!card.init(SPI_HALF_SPEED, 4)) return;
SdFile file;
//opening file with O_READ constant
if (!file.open(fileName, O_READ)) return;
```

---

There are more constants available wich allows the programmer to append lines to the texts or to truncate the whole text.

### Writing/Reading files

---

#### Algorithm 18 Arduino Writing/Reading Files

---

```
char buffer[255];
//fills the buffer with the line
file.fgets(line, sizeof(line));

//writes the a line to the file
file.println("sampleString");
```

---

### Closing files

---

#### Algorithm 19 Arduino Closing Files

---

```
file.close();
```

---

### 3.3.2 MCP Arduino

The MCP Handling is done in several classes shown in this diagram.

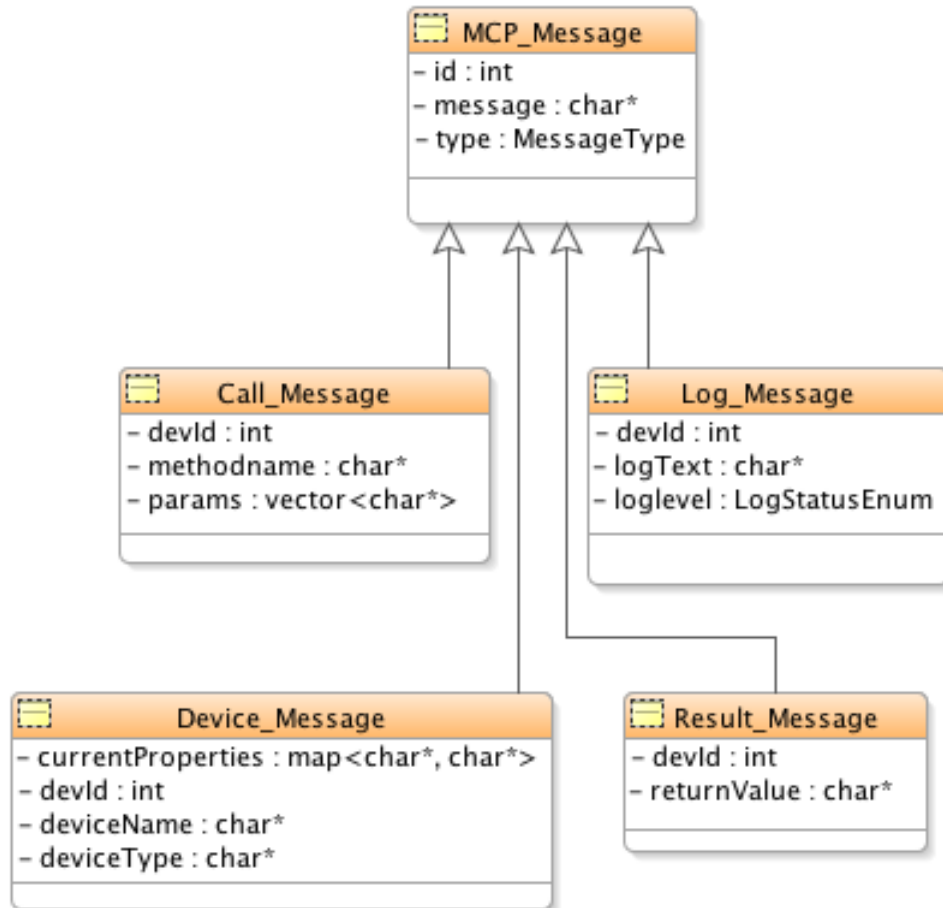


Figure 16: MCP Class Diagram Arduino

The base class of all messages is **MCP\_Message** which defines the `MessageId` as well as the `MessageType` which is in every `MessageType` the same. When the Constructor of the derived classes is called, this class calls at first the base constructor to parse the `MessageId` and the `MessageType`. Then further processing is done using the properties of the classes.

Messages are received in the **Connector** class. With every program `loop()` iteration, new messages are checked through the **MessageObserver** class. This class has the method `CheckforMessages()` which is called in the main loop. In the constructor of the Mes-



sageObserver class, the observer signs up for his callback method using the following code:

---

**Algorithm 20** Arduino MessageObserver Initialization

---

```
//retains an instance of the Connector singleton class
this->connector = Connector::getInstance();
//defines that observing serial connections should be disabled
this->observeSerial=false;
//defines that observing ethernet connections should be disabled
this->observeEthernet=false;
//initialises the serial connection
connector->initSerial();
//initialises the ethernet connection
connector->initEthernetShield();
//retains an instance of the MCP_Interpreter singleton class
interpreter = MCP_Interpreter::getInstance();
//retains a new MCP_MessageListener object
listener = new MCP_MessageListener();
```

---

The Connector class was built after the Singleton pattern because there has to be only one instance of this class in the whole system. The private properties observeSerial and observeEthernet determine which type of connection has to be used. The Interpreter class is also a Singleton because there must be one instance of this class. Messages were processed consecutive one after another and not at the same time. MessageListener is the class which holds the &OnMessageReceived callback.

Once the object exists, the subscribeListener method can be called to register the callback in the Connector class. The following code will be executed:

---

**Algorithm 21** MessageObserver SubscribeListener

---

```
void MCP_MessageObserver::subscribeListener()
{
    this->connector->setMessageCallback(listener,&MCP_MessageListener::
        OnMCPMessageRecieved);
}
```

---

In the main loop the checkForMessages() method will be called in every iteration.

In the background, the Connector class continuously checks if messages are available. If a message has been received successfully, the callback is called:

---

**Algorithm 22** MessageListener OnMCPMessageReceived

---

```
void MCP_MessageListener::OnMCPMessageReceived(char* msg)
{
    //calls the interpret method and retains the result
    char* result = MCP_Interpreter::getInstance()->interpret(msg);
    //sends the result back
    Connector::getInstance()->sendResult(result);
}
```

---

The callback then calls the MCP\_Interpreter instance which interpretes the message using the defined format. It splits the messages and depending on the type calls the functions. Received types can be LogMessages and CallMessages. The process of receiving messages and checking them is described beneath.

Once the CallMessage class exists representing the string message received, the device is selected by the DeviceManager class and the method is called dynamic. The process of calling the methods is described in 3.3.4

If there is a result from the method, this result is sent back by the connector class. The connector class keeps the last address used in memory and sends it back without the need of specifying the IP-address or the COM port. This connection is stored in the Connection class determining which ConnectionType is used and also which IP-address if needed. ConnectionType is an enum defining the types ETHERNET, COM or ADK.

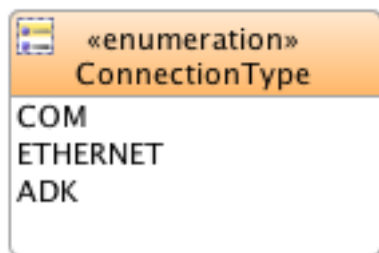


Figure 17: Connection Type

### 3.3.3 MCP Java

#### MCP messages

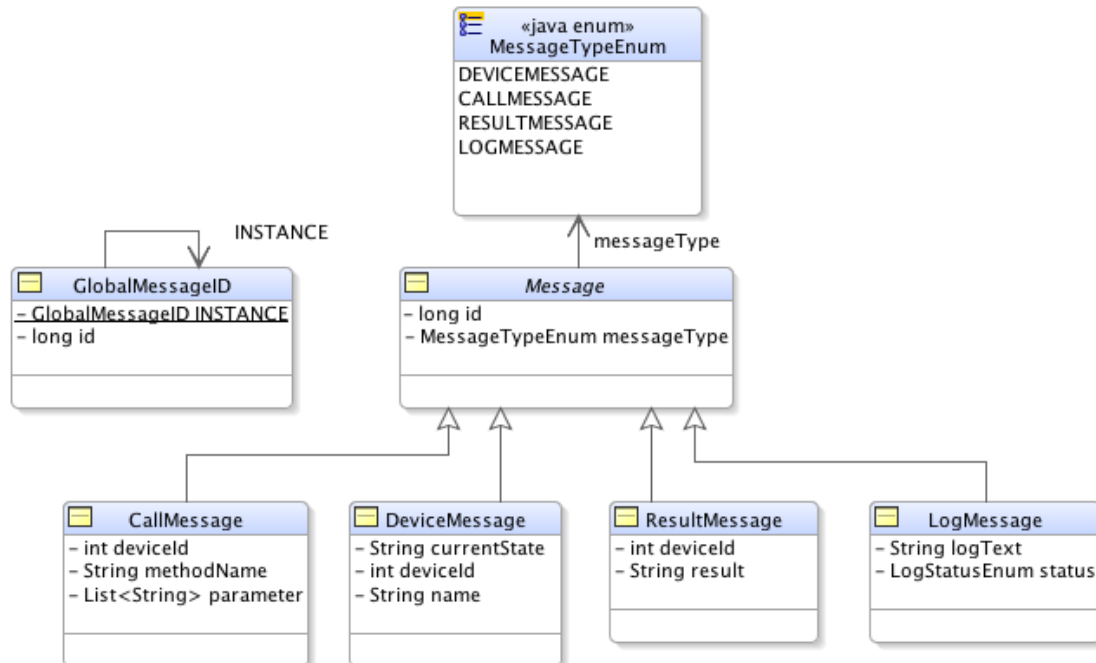


Figure 18: MCP message overview

All message types (`CallMessage`, `DeviceMessage`, `ResultMessage`, `LogMessage`) have a abstract base class `Message`. The base class `Message` has an `id` which is set through the singleton `GlobalMessageID`. It also contains a `MessageType`, which is an enum value, therefore the enum contains the four message types.

### DeviceMessage

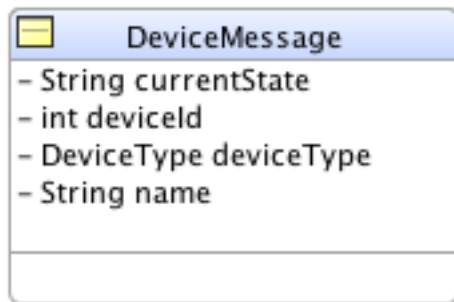


Figure 19: DeviceMessage class

Every time when the server starts it receives all devices. To make that possible the DeviceMessage was implemented and therefore it represents a device. It has the same properties as a device except for the IP - address which it gets from the socket.

### CallMessage

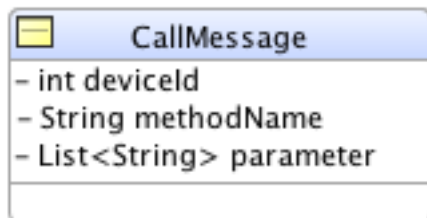


Figure 20: CallMessage class

A CallMessage will be send to the Arduino MC for calling a method. The deviceId is used to determine the corresponding device and the methodName is the method which will be called. If the CallMessage is global then a '0' will be used a deviceId, for example the method 'getDevices'. If the method has any arguments then they are in the list of string parameter.

### ResultMessage



Figure 21: ResultMessage class

Every time when the server sends a `CallMessage` to the Arduino MC it will send a `ResultMessage` back. Again the `deviceId` is used to determine the corresponding device and if it's global '0' is used. The string `result` contains the result which can be 'true' if it's only sent as an acknowledgment of receipt or something like 'temperature=21' if it's the value from a sensor.

### LogMessage

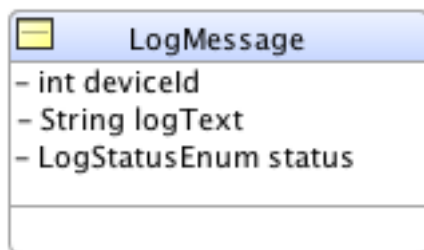


Figure 22: LogMessage class

The `LogMessage` is used to log events, errors, etc. The `logText` contains the message what happened and the `status` is a value of an enum which can take the values 'ERROR', 'WARNING' or 'NORMAL'. The `deviceId` is used to determine the corresponding device and if it's global '0' is used.

## GlobalMessageID



Figure 23: GlobalMessageID class

The `GlobalMessageID` is a singleton class and it's used to distribute the ids to the messages. If a message is created it will automatically get an id from this class.

## MessageTypeEnum

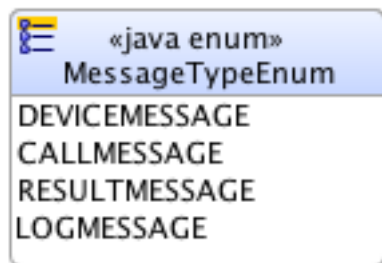


Figure 24: MessageTypeEnum

This enum contains all message types. The message type is sent within the string which the socket receives and that's why the server is able to create the right message out of this string.

## MCP communication

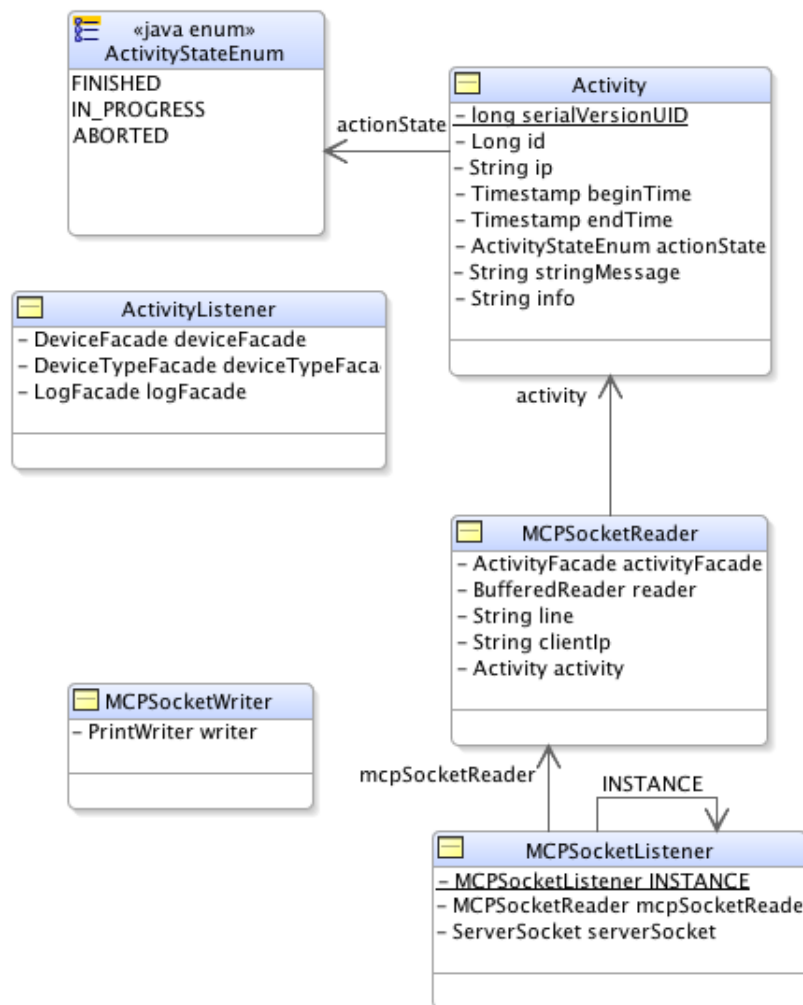


Figure 25: MCP communication overview

The communication between the server and an Arduino MC begins at the bottom right corner of the diagram (25). The MCPSocketListener class has a SocketServer which listens on port 13316 and of course the class runs in a separate thread. When this class gets a message, it creates a new MCPSocketReader, also in a separate thread.

Now the MCPSocketReader gets/reads all messages from the socket and creates for each message an activity. The activity class is made for statistic usages, but it's main aspect is for creating the message from the stringMessage through the ActivityListener class.

---

**Algorithm 23** Java EE Server - MCPSThreadReader

---

```
//Reads the first line from the socket
line = reader.readLine();

while (line != null && reader != null) {
    //Sets the GlobalMessageID from the incoming message
    GlobalMessageID.getInstance().setId(Long.parseLong((line.substring(1)).
        split(":")[0]));
    //Creates a new activity with the ip from the sender, the beginTime as
    //timestamp, sets that the activity is in progress and the
    //stringMessage
    activity = new Activity(clientIp, new Timestamp(new Date().getTime()),
        ActivityStateEnum.IN_PROGRESS, line);
    //Persists the activity
    activityFacade.create(activity);
    //Reads the next line
    line = reader.readLine();
}
//If the reader isn't closed yet, then the reader will be closed
if (reader != null) reader.close();
```

---

Before the activity gets persisted into the database, the method 'prePersistActivity(Activity activity)' in the ActivityListener class gets called because of the @EntityListeners(ActivityListener.class) annotation in the Activity class and the @PrePersist annotation in the ActivityListener class.

In the ActivityListener class the stringMessage will be casted to a CallMessage, ResultMessage, LogMessage or DeviceMessage. If it's a DeviceMessage then a device will be created and persisted into the database. If the stringMessage is a ResultMessage the corresponding device gets it's state updated. If it's a LogMessage then a Log will be created and will get persisted into the database. And finally if it's a CallMessage nothing happens, because the server shouldn't receive one. The activity info property gets updated accordingly in all situations.



---

**Algorithm 24** Java EE Server - ActivityListener

---

```
@PrePersist
private void prePersistActivity(Activity activity) {
    //Getting the Facades

    //Converts the stringMessage to a Message
    Message message = getMessage(activity.getStringMessage());

    if (message != null) {
        Device device;
        //Check which message type it is
        switch (message.getMessageType().ordinal()) {
            case 0: //DeviceMessage
                //A device will be created and persisted into the database
                //The activity will be set to finished, an endtime will be set and
                //the info will be updated
                break;
            case 1: //CallMessage
                //The server shouldn't receive a CallMessage, therefore the activity
                //will be set accordingly
                activity.setEndTime(new Timestamp(new Date().getTime()));
                activity.setActivityState(ActivityStateEnum.ABORTED);
                activity.setInfo("RCSServer shouldn't get a CallMessage.");
                break;
            case 2: //ResultMessage
                //The state of the corresponding device will be updated
                //The activity will be set to finished, an endtime will be set and
                //the info will be updated
                break;
            case 3: //LogMessage
                //A Log will be created and persisted into the database
                //The activity will be set to finished, an endtime will be set and
                //the info will be updated
                break;
        } else {
            activity.setEndTime(new Timestamp(new Date().getTime()));
            activity.setActivityState(ActivityStateEnum.ABORTED);
            activity.setInfo("Invalid message!");
        }
    }
}
```

---

The MCPSocketWriter is created when the server sends a message. The message will be send in a separate thread, because the destination may be offline and this causes a exception and if it wouldn't be sent in a separate thread the whole program would stop for the duration of a timeout.

---

**Algorithm 25** Java EE Server - MCPSocketWriter

---

```
public class MCPSocketWriter extends Thread {
    private String ip;
    private Message message;

    public MCPSocketWriter(String ip) {
        this.ip = ip;
    }

    public void send(Message message) {
        this.message = message;
        this.start();
    }

    @Override
    public void run() {
        try {
            //Creates the socket to the destination
            Socket socket = new Socket(InetAddress.getByName(ip), 13316);
            //Creates the writer
            PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
            //Stops for 20ms because if messages are sent without timeout, then
            they may be lost
            Thread.sleep(20);
            //Sends the message
            writer.println(message.toString());
            //Closes the writer and socket
            writer.close();
            socket.close();
        } catch (Exception ex) {
            System.out.println("MCPSocketWriter.send: " + ex.toString());
        }
    }
}
```

---

### 3.3.4 Control Apps Arduino

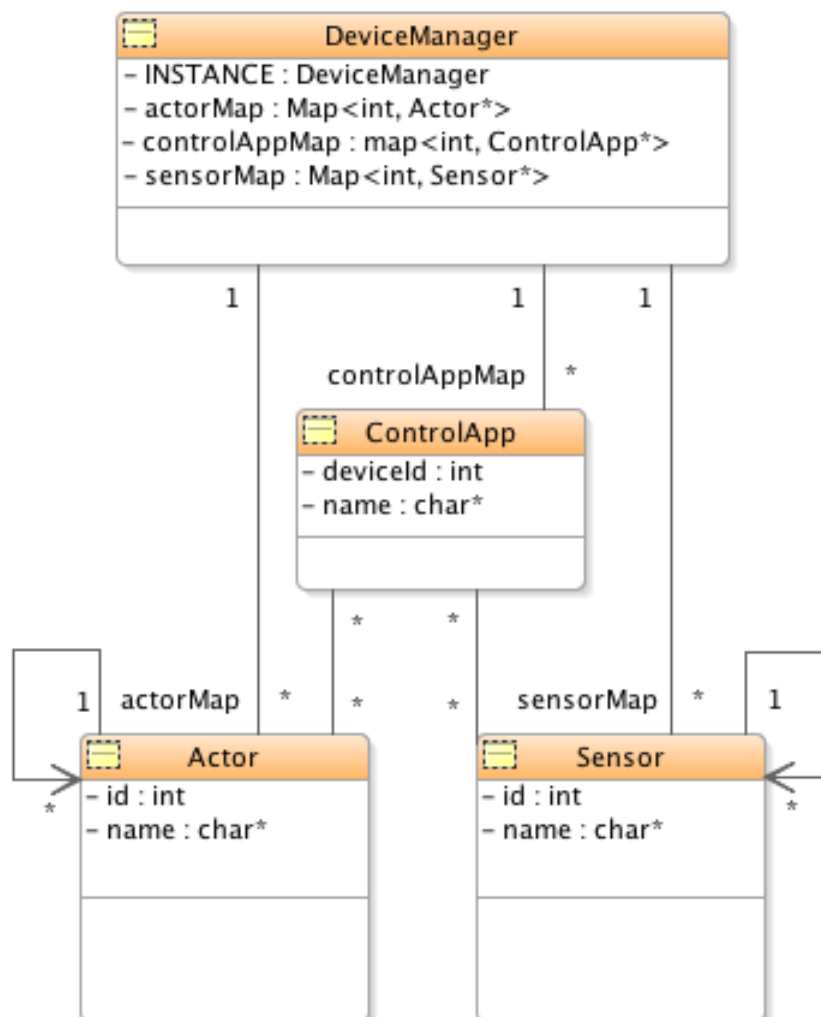


Figure 26: Device Structure Arduino

**Control App Structure** Every **ControlApp** consists of 0 to n **Actors** and 0 to n **sensors**. **Actor** classes generally offer methods to invoke actions, while **sensors** generally are used to monitor the environment conditions. **ControlApps** are the interface to the Arduino part. They offer methods which can be called from outside the Arduino system different from **actors** and **sensors** which can only be invoked within the system. The deeper the **actor** or the **sensor** is in the hierarchy, the more hardware based is it. This simple example shows how this paradigm can be used.

Every ControlApp, every sensor and every actor are administered by the DeviceManager Singleton Class. Any app must derive from the base class named ControlApp. In these class, general methods are defined which have to be implemented. They were needed for communicating with the environment. These methods are:

---

**Algorithm 26** Arduino Control App Methods

---

```
//needed for calling a method by passing its name and parameters
virtual char* callMethod(String methodName, std::vector<char*> params) =
    0;
//needed for parallel working
virtual bool doWork() =0;
//needed for sending all relevant properties
virtual std::map<char*,char*> getProperties()=0;
```

---

**Calling a method using jumptables** Due to the fact that we are working on a very low level hardware and software system, reflection is not available. Generally in C++ reflection is available with different Libraries such as RTTI (Run-Time Type Information). Porting would hardly be possible because of the limited resources on the Arduino system. So instead of dynamically calling the method, there is a little workaround which we implemented. Every Control App has to implement a method named callMethod accepting two parameters, the name of the method to be called and the params. How this method is implemented inside, does not matter, but in fact, there is only one solution which suits best for this operation. Making a jumpTable improves the way a method is called compared to simple if statements. Once such a table is implemented, only the methodname and the pointer to the method has to be added. Here is an example implementation of such an method:

---

**Algorithm 27** Arduino CallMethod

---

```
char* SensorDevice::callMethod(String methodName, std::vector<char*> params)
{
    for(int i = 0; i<GetArrayLength(functionNames); i++)
    {
        if(functionNames[i] == methodName)
        {
            return (this->*functionArray[i])(params);
        }
    }
    return METHOD_NAME_NOT_FOUND;
}
```

---

The code above iterates through the functionNames array and if it finds the same name as the one passed as a parameter, it calls the appropriate method using the jumpTable named functionArray. In the class file, the following example definitions have been made:

---

**Algorithm 28** Arduino JumpTable

---

```
//defines a new type named ptMember for easier future use
typedef char* (SensorDevice::*ptMember)(std::vector<char*>);
//initialises the jumpTable
ptMember functionArray[3] = {&SensorDevice::getLuminosity,&SensorDevice::
    getTemperature,&SensorDevice::getMovementRaw};
//initialises the appropriate method names
char* functionNames[3] = {"getBrightness","getTemperature","getMovement"};
```

---

**Multitasking on Arduino hardware** Due to the fact that multitasking is not available on the Arduino board, we implemented a simple but effective method which allows all Control Apps to do their work at nearly the same time if needed. The idea behind the concept is that every class which wants to run simultaneously has to implement the doWork Method. Internally, this method has a certain entry point where data are processed. Using this structure a cooperative multi tasking system is established. At any time while the system is running, Apps can add themselves to a working queue using the following code:

---

**Algorithm 29** Arduino WorkQueue Add Elements

---

```
WorkQueue::getInstance()->add(this);
```

---

In this working queue, all the doWork methods of the subscribed apps were called continuously one after another. The only thing which must not be done by any app is the use of blocking functions such as sleep. If, for example the app wants to run through some lines of code every one second, it has to use the millis function which is defined by the Arduino framework. Using this technique, the doWork methods are called without much delay. The implementation of the WorkQueue class allows users to remove themselves at any time they want by returning false in the doWork method or by calling WorkQueue::remove(this);

**Getting all the relevant properties for simple data exchange** At the client, on startup, every property has to be retrieved by sending Callmessages. In further consequence, the UI system may doesn't react any more because sending, computing and retrieving messages costs time. So we defined that every Control App also has to have the availability to retrieve all data from the device at once. By default, when all the available devices are sent, this information is also sent. It is implemented using the Method getProperties. It returns a map of variables which are relevant for the UI. An example implementation could be as follows:

**Algorithm 30** Arduino GetProperties

```

std::map<char*,char*> SensorDevice::getProperties()
{
    //allocate a new map called props
    std::map<char*,char*> props;
    //allocating a new charArray named temp with 20 characters
    temp = (char*) malloc(20);
    //filling the free char* with the current temperature
    sprintf(temp, "%s", this->getTemperature());
    //filling the map
    props["temperature"] = temp;
    //returning the map object
    return props;
}

```

**Sending Control App information** When the client starts up, it does not know how many devices are available or what devices these are. So we implemented a global method called `getDevices` which is able to iterate through all devices and send back their specific information.

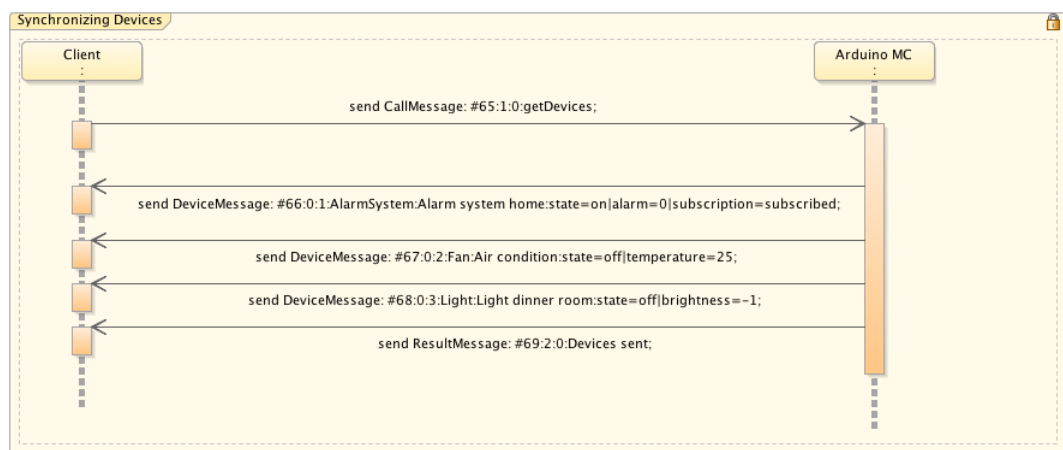


Figure 27: Synchronizing Devices

When the `CallMessage` is sent, a number of `DeviceMessages` are sent as an information. The last messages is in any case a `ResultMessage` determining if the transfer was successful or not. Information concerning `DeviceMessages` can be found at 3.2.2.

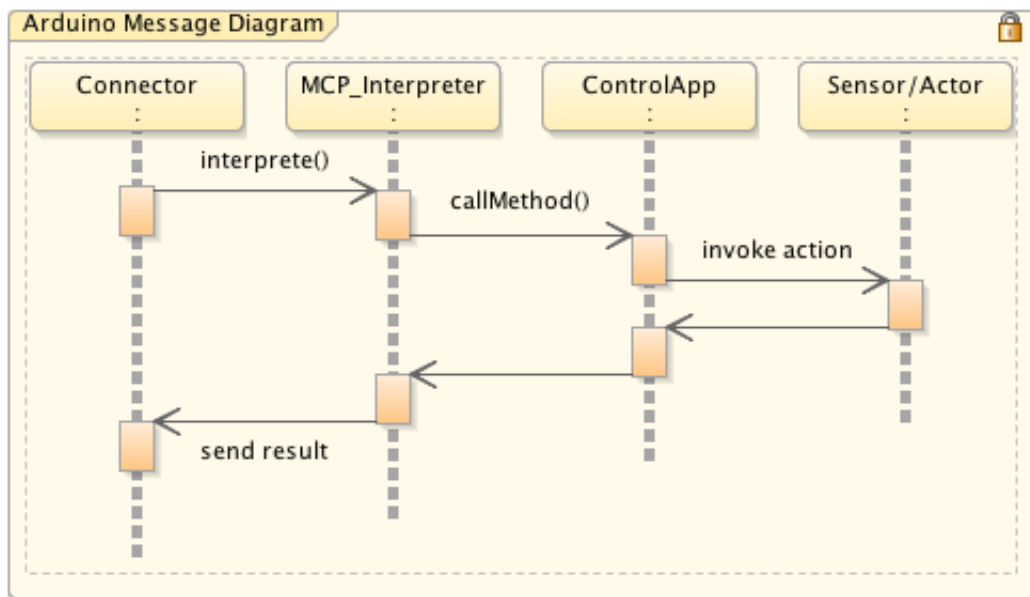


Figure 28: Arduino Message Diagram

**Flow Diagram** Once a message has been received, the Singleton class `MCP_Interpreter` is called. This class basically contains the handling of messages in runtime. The `interpret` method returns the result received from the different `ControlApps` or the exception messages when for example a device was not found. At first, this method splits the incoming message, proves if the message is a `CallMessage` and then it creates a `Call_Message` instance based on these data.

---

**Algorithm 31** Arduino MessageType Checking
 

---

```

std::vector<char*> elems;
//splitting the recieved charArray and filling the vector
StringUtility::split(mcpMessage, ":", elems);
char t = elems[1][0];
//parsing the messageType and checking if it is a callmessage
MCP_Message::MessageType type = (MCP_Message::MessageType) (t - '0');
if (type == MCP_Message::Call)
{
    //method for handling CallMessages
    return this->handleCallMessage(elems);
}
  
```

---

Now the retrieved `deviceId` is proved and when 0 is passed, global methods were executed. If a `DeviceId` greater than 0 is passed, the `DeviceManager`'s method `getDeviceById` is

called with the deviceId as Parameter. If the Device is found, the callMethod function is called which is described above. It returns a result which is then immediately passed back and sent to the requesting instance.

---

**Algorithm 32** Arduino Calling Device Methods

---

```
//retaining the appropriate device
Device *dev = DeviceManager::getInstance()->getDeviceById(callmsg.getDevId());
//calling the method by its name and returning the result
char* erg = NULL; return dev->callMethod(callmsg.getMethodName(), callmsg.getParams());
```

---

Note that the code above is simplified to show the general procedure.

**Storing and Saving** Due to the fact that after every shutdown control apps, actors and sensors have to be loaded again on the next startup, we implemented a system used for persisting control apps, actors and sensors. Saving of classes was a straightforward matter, but loading them again without the use of reflection was far more complicated. We implemented a reflection like system for generating instances of classes using templating and factory design patterns. We created one Singleton class named UnifiedFactory which abstracted the loading of classes. In order to implement a method named generateClassFromName, we also implemented an interface named IPersistable which every control app, actor and sensor should implement. It is defined as followed:

---

**Algorithm 33** Arduino IPersistable Interface

---

```
class IPersistable {
public:
    virtual std::map<char*,char*> save()=0;
    virtual IPersistable* load(std::map<char*,char*,str_cmp>)=0;
    virtual int getId()=0;
    virtual char* getClassName()=0;
};
```

---

When implementing this interface, every class has its own class name which can be retrieved by calling getClassName. This allows us to choose which template object should be instantiated. Templates were instantiated with the following line of code:

---

**Algorithm 34** Arduino Template Definition

---

```
IPersistable* persistableClassArray[9] = { new SensorDevice(), new
SolidStateRelay(), new PowerOutlet(), new Heater(), new
TemperatureSensorPhidget(), new PrecisionLightSensorPhidget(), new
Light(), new Rolladen(), new Fan()};
```

---



It makes no difference whether these classes are actors or sensors, it only bothers that they implement the IPersistable interface. When generateClassFromName with the method-name and the instancing parameters is called, the system iterates through the array and checks if this class is available for instantiation. If the class is found, the load method is called. The load method must not return a pointer to its own object because the objects instantiated here are only templates which must not be able to live outside this scope. The generateClassFromName function is defined as follows:

---

**Algorithm 35** Arduino GenerateClassFromName

---

```
Persistable* UnifiedFactory::generateClassFromName(char* className, std::map<char*,char*,str_cmp> params);
```

---

If the object is successfully referenced, it is passed back to the DeviceManager class. Otherwise NULL will be passed back.

An example implementation of load would be:

---

**Algorithm 36** Arduino Template Generation

---

```
//Generating a new object of type Rolladen
Rolladen* rolladen = new Rolladen(StringUtility::atoi(map["position"]));
//Setting the id with the map which was passed as a parameter
rolladen->setId(StringUtility::atoi(map["id"]));
//Setting the SSR by retrieving it from the DeviceManager singleton class
rolladen->setSSRUp((SolidStateRelay*)DeviceManager::getInstance()->
    getActorById(StringUtility::atoi(map["ssrUpId"])));
//Setting the SSR by retrieving it from the DeviceManager singleton class
rolladen->setSSRDown((SolidStateRelay*)DeviceManager::getInstance()->
    getActorById(StringUtility::atoi(map["ssrDownId"])));
return rolladen;
```

---

The StringUtility class offers basic methods for working with char arrays, for example atoi which converts char arrays to integer. The actors needed for the rollerblind are retrieved by calling getActorById with the saved Id of the actor. The actor itself is not saved, instead the id of it is saved and loaded then again using the DeviceManager class. The device itself must ensure that the actor exists and is properly loaded.

There are two methods which handle the saving and the loading of all actors, sensors and ControlApps in the Devicemanager class called writeToSdCard and readFromSdCard accepting no arguments.

### readFromSdCard

The readFromSdCard Method calls the private load method with three different parameters as shown below:

---

**Algorithm 37** Arduino Calling the Load Method

---

```
if(-1==this->load("Actors.mcp",&actorMap)) return -1;
if(-1==this->load("Sensors.mcp",&sensorMap)) return -1;
if(-1==this->load("Devices.mcp",&deviceMap)) return -1;
```

---

the load method basically opens the file which is passed as a parameter. It then iterates through the map passed as a reference. Due the fact that actors, sensors and ControlApps are all implementing the IPersistable Interface, the load method casts the passed map to a persistable map using the following syntax:

---

**Algorithm 38** Arduino Casting IPersistable Map

---

```
std::map<int, IPersistable*, ltDev>* persistableMap = static_cast<std::map<
    int, IPersistable*, ltDev>*>(ptr);
```

---

Static casts are casts between pointers of the same hierarchy. It isn't ensured that the cast is successful, but one parameter must derive the other parameter. In our case, Actor\* derives from IPersistable. Then, the lines are read and the object is generated using the UnifiedFactory class.

**writeToSdCard**

The writeToSdCard method calls the private persist method with different parameters:

---

**Algorithm 39** Arduino Calling the Persist Method

---

```
if(-1==this->persist("Actors.mcp",&actorMap)) return -1;
if(-1==this->persist("Sensors.mcp",&sensorMap)) return -1;
if(-1==this->persist("Devices.mcp",&deviceMap)) return -1;
```

---

The persist method casts the pointer with an static\_cast to a persistable map and iterates through the map. Every element's save method is called and the result is concatenated to one string which is saved afterwards. The file which is passed by the parameter is opened and every device is saved.

**Store Format**

The different IPersistable Classes are stored in a simple format with the following structure:

```
<ClassName>
<parameterKey>=<parameterValue>|<parameterKey>=<parameterValue>|<parameterKey>
```

**Steps for developing own Control Apps** First of all, a new class has to be created as an Android library. That means that the containing folder is named identically to the class which has to be created and the folder is in the libraries path which can be customized with the Arduino IDE. Once the class is created, it has to be derived from the ControlApp base class and the following methods have to be declared and implemented:

---

**Algorithm 40** Arduino ControlApp Methods

---

```
virtual char* callMethod(String, std::vector<char*>) = 0;  
virtual bool doWork() =0; virtual std::map<char*,char*> save()=0;  
virtual IPersistable* load(std::map<char*,char*,str_cmp>)=0;  
virtual char* getClassName()=0; virtual std::map<char*,char*> getProperties  
    ()=0;
```

---

How the different methods are implemented is described above.

The second step is that once the implementation is finished to add this class to the Unifiedfactories perstistableClassArray properties.

### 3.3.5 Devices Java Side

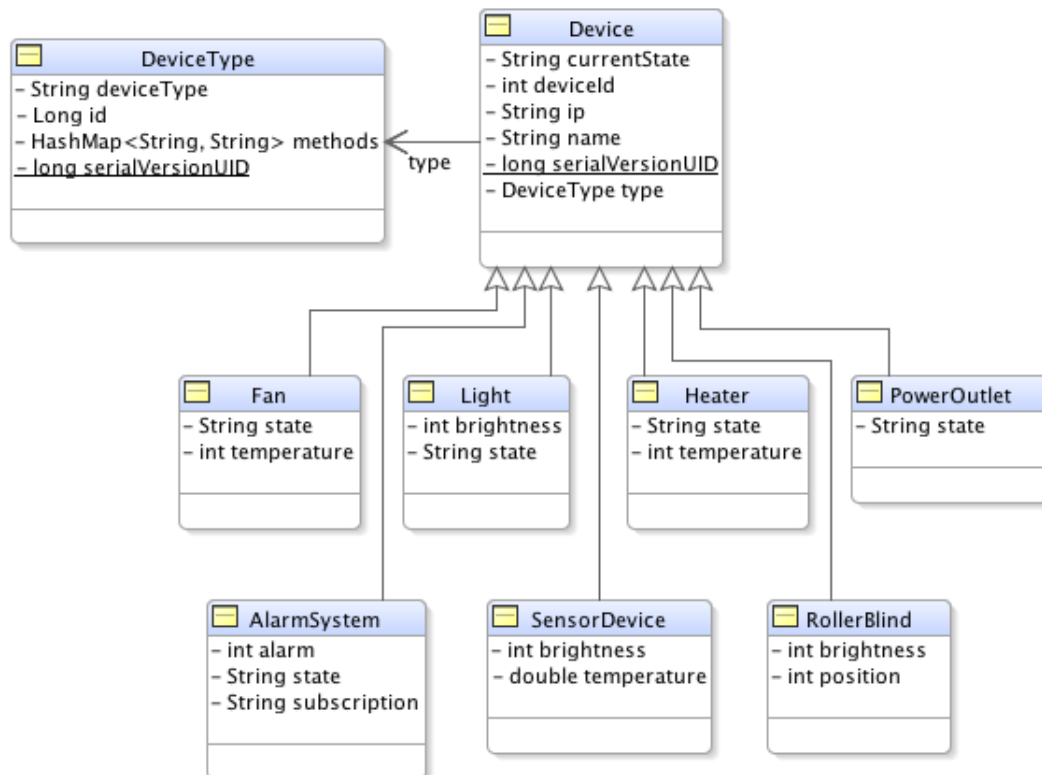


Figure 29: Devices overview

The figure 29 shows how the devices are connected. In the following pages every class with it's structure is described.

## DeviceType



Figure 30: DeviceType class

The DeviceType contains the device type as a string, for example „RollerBlind“, etc. It is essential, that this string equals the class name, because in the UI the form for each device is loaded with this string. It also contains a HashMap for the methods, which should be available for the tasks, explained later. The public methods are the constructors and the getters and setters for the properties.

## Device

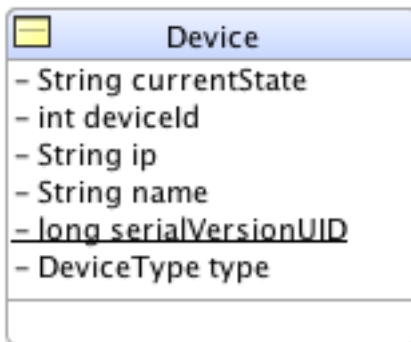


Figure 31: Device class

The device class is the base class for each device. Normally it should be abstract, but it isn't so because this class is used to persist them into the database and therefore it's

needed to instantiate it. Each device has an unique id, an IP - Address, a name and a device type. Each device which is derived from this class has it's own states and they are persisted in this class as currentState.

### AlarmSystem

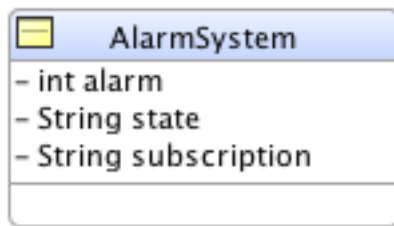


Figure 32: AlarmSystem class

The AlarmSystem has an alarm as integer with the value '0' or '1', which represents if it is activated or not. The state can take the values 'on' or 'off' and therefore represents if the AlarmSystem is turned on or turned off. The subscription declares if the server will receive an alarm or not and can take the values 'subscribed' or 'unsubscribed'.

### Fan

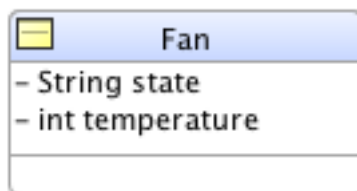


Figure 33: Fan class

The integer value temperature is responsible for the temperature condition, which means that if the temperature of the sensor exceeds this value the fan will be turned on. The state can take the values 'on' or 'off' and therefore represents if the fan is turned on or turned off.

## Light



Figure 34: Light class

The data structure of the light is nearly the same as the structure of the fan. The only difference is, that the light doesn't use a temperature but a brightness for it's condition.

## Heater

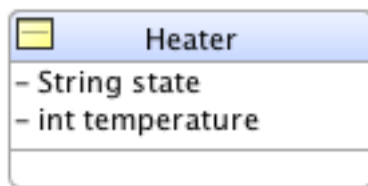


Figure 35: Heater class

The data structure of the heater is equivalent to the data structure of the fan.

## PowerOutlet

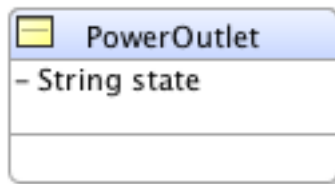


Figure 36: PowerOutlet class

The only property that a PowerOutlet has, is a state, which can take the values 'on' or 'off'.

## SensorDevice

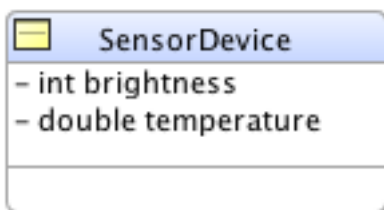


Figure 37: SensorDevice class

The SensorDevice represents a light sensor and a temperature sensor. It can't do any actions and only shows the temperature and the brightness of the sensors.



## RollerBlind

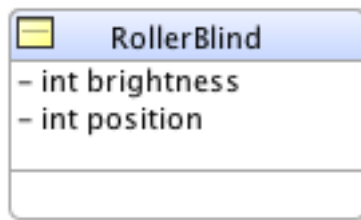


Figure 38: RollerBlind class

The RollerBlind has got a variable position for his current physical position. The brightness represents the brightness condition and if the brightness of the sensor falls below this condition the RollerBlind moves down.

### 3.3.6 Tasks & Profiles

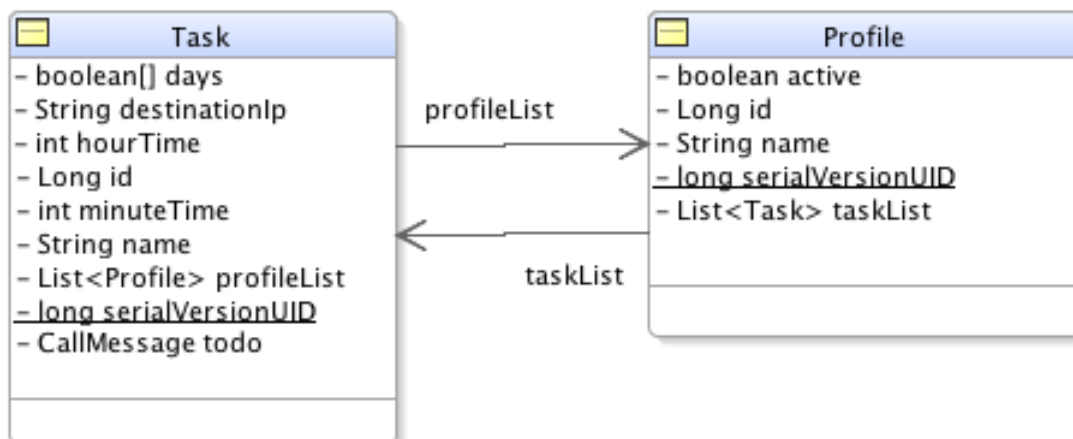


Figure 39: Profile-Task overview

To reach a certain automation in our system profiles and tasks are used.

For example profiles are 'Holiday profile' or a 'Work profile'. These profiles can be activated (boolean active) and they have a list of tasks which should be processed if it is active. This happens in the ProfileScheduler:

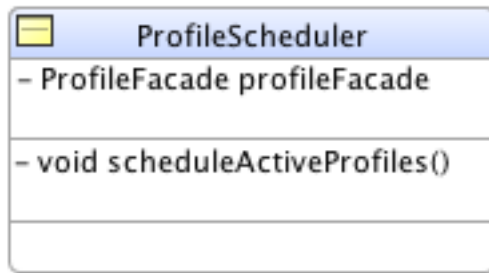


Figure 40: ProfileScheduler

The `ProfileScheduler` contains the method `'scheduleActiveProfiles()'` with the `@Schedule(hour = "*", minute = "*/1", persistent = true)` annotation, which means that it every hour every single minute this method is called. What happens in this method you can see below in the code:

---

**Algorithm 41** Java EE Server - ProfileScheduler

---

```
@Schedule(hour = "*", minute = "*/1", persistent = true)
private void scheduleActiveProfiles() throws ParseException,
    InterruptedException {
    //Retrieve all profiles from the database
    List<Profile> profileList = profileFacade.findActiveProfiles();

    if (profileList != null && !profileList.isEmpty()) {

        for (Profile profile : profileList) {
            //Get the current time
            Timestamp timestamp = new Timestamp(Calendar.getInstance().getTime().
                getTime());
            //Retrieve all tasks from the current profile
            List<Task> taskList = profile.getTaskList();

            for (Task task : taskList) {
                //Check if the current time and day equals with these from the task
                if (task.getDays()[timestamp.getDay()] == true
                    && task.getHourTime() == timestamp.getHours()
                    && task.getMinuteTime() == timestamp.getMinutes()) {

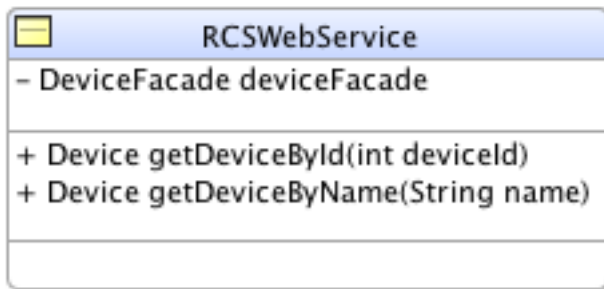
                    //Create a new MCPSocketWriter with the destination of the task
                    MCPSocketWriter writer = new MCPSocketWriter(task.
                        getDestinationIp());
                    //Send the CallMessage
                    writer.send(task.getTodo());
                }
            }
        }
    }
}
```

---

The task also has to know to which profile it belongs, because it mustn't be deleted if a profile uses it.

### 3.3.7 SOAP Webservice

The SOAP web service is built up quite simple:



CallMessage

Figure 41: SOAP WebService

In the code below you are able to see the RCSWebService class:

---

**Algorithm 42** Java EE Server - SOAP Webservice

---

```
@WebService(serviceName = "RCSWebService")
public class RCSWebService {

    @EJB
    private DeviceFacade deviceFacade;

    @WebMethod(operationName = "getDeviceByName")
    public Device getDeviceByName(@WebParam(name = "name") String name) {
        return deviceFacade.findByName(name);
    }

    @WebMethod(operationName = "getDeviceById")
    public Device getDeviceById(@WebParam(name = "deviceId") int deviceId) {
        return deviceFacade.findbyDeviceId(deviceId);
    }
}
```

---

As you can see this code is really simple, it only returns the requested device. And at the client you have to use the `@WebServiceRef` annotation on the device because this makes it possible to invoke a SOAP web service.

*„With the WSDL and some tools to generate the Java stubs, you can invoke a web service. Invoking a web service is similar to invoking a distributed object with RMI. Like RMI, JAX-WS enables the programmer to use a local method call to invoke a service on another host.“ [8]*

### 3.3.8 Web Interface

The web interface is built up into tabs, which are separated in a Devices, Profile & Task, Settings and Log tab.

#### Devices

In the devices tab the devices are listed in a table as shown below:

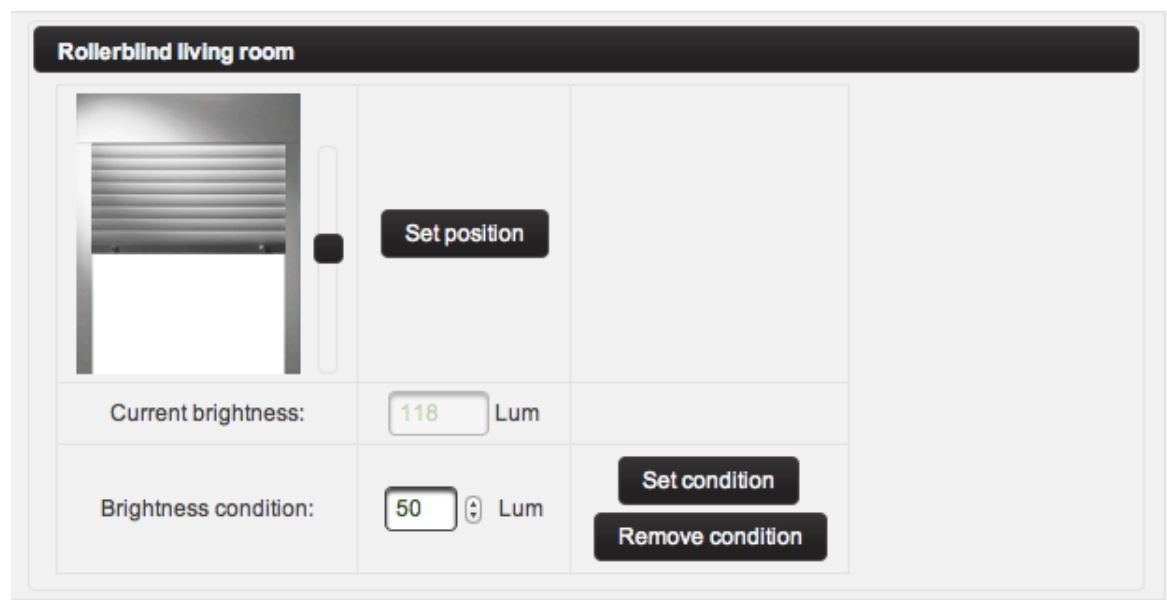


Name	Type	IP-Address
Light dinner room	Light	Local (192.168.1.2)
Air condition	Fan	Local (192.168.1.2)
House electricity	PowerOutlet	Local (192.168.1.2)
Rollerblind living room	RollerBlind	Local (192.168.1.2)
AlarmSystem	AlarmSystem	Local (192.168.1.2)
Heater	Heater	Local (192.168.1.2)


Figure 42: Devices table

Each device in this table is selectable and if a device is selected the region beside the table will be updated.

#### RollerBlind



**Rollerblind living room**



**Set position**

Current brightness:  Lum

Brightness condition:  Lum

**Set condition**

**Remove condition**

Figure 43: Webinterface RollerBlind

In the first row the rollerblind position can be set with the slider and the button. The second row shows the current brightness. In the third row the user is able to set/remove the brightness condition of the rollerblind.

The rollerblind is made with a primefaces slider and this slider is connected per JavaScript/CSS/HTML to the rollerblind image beside it. The code below shows how this exactly works:

---

**Algorithm 43** Java EE Server - RollerBlind JavaScript

---

```
function rollerblind_moveToPosition(sliderTextId , imgInsideId) {  
    //Gets the inner image of the rollerblind  
    var imgInside = document.getElementById(imgInsideId);  
    //Gets the hidden inputtext for the slider  
    var sliderText = document.getElementById(sliderTextId);  
  
    //Calculates the position at which the image should be  
    var position = parseFloat(1000 - sliderText.value);  
  
    //Calculates the the highest position that the image may be  
    var max = parseFloat(parseFloat(imgInside.height) * 19 / 20 * (-1));  
  
    //Looks if the position is in the expected range  
    if (position <= 1000 && position >= 0) {  
        //Sets the top position of the inner image of the rollerblind  
        imgInside.style.top = max * (1 - (position / 1000)) + 'px';  
    }  
}
```

---

This is the JavaScript part of the rollerblind. This is a method for positioning the inner image of the rollerblind, so that the image is correctly positioned to the value of the slider.

**Algorithm 44** Java EE Server - Webinterface RollerBlind

---

```
...
<div id="rollerblind#{deviceController.selectedDevice.deviceId}"
class="standard"
style="width: 140px; height: 150px; position: relative;">
  <div id="divOutside#{deviceController.selectedDevice.deviceId}"
style="position: relative; width: 120px; height: 150px; float: left;">
    
    <div id="divInside#{deviceController.selectedDevice.deviceId}"
style="position: relative; overflow: hidden; top: 27px; left: 8px;
width: 104px; height: 124px;">

      //When the image is loaded the JavaScript method will be called and
      the right position of this image will be set
      
    </div>
  </div>
<div style="position: absolute; bottom: 0; right: 0px;">

  //Whenever the slider is used the JavaScript method is called and
  therefore the position of the image will be set
  <p:slider id="slider#{deviceController.selectedDevice.deviceId}"
for="sliderText#{deviceController.selectedDevice.deviceId}"
type="vertical"
minValue="0"
maxValue="1000"
style="height: 120px; float: right, bottom;"
onSlide="rollerblind_moveToPosition(
  'indexTabView:devicesForm:sliderText#{deviceController.selectedDevice
    .deviceId}',
  'imgInside#{deviceController.selectedDevice.deviceId}');"
onSlideEnd="rollerblind_moveToPosition(
  'indexTabView:devicesForm:sliderText#{deviceController.selectedDevice
    .deviceId}',
  'imgInside#{deviceController.selectedDevice.deviceId}');" />
  </div>
</div>
...
```

---

This is the HTML/(CSS) part of the rollerblind. As you can see the CSS is hardcoded into the style propertie of the elements. The JavaScript method of the rollerblind is once called, when the image is loaded and again if the position of the slider is changed.

## Fan

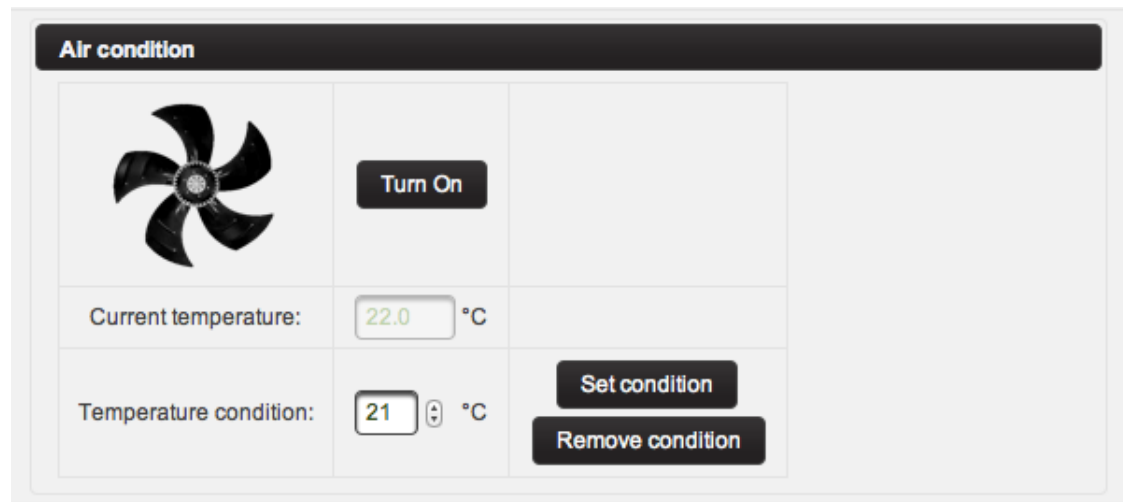


Figure 44: Webinterface Fan

In the first row a image of a fan is shown and if it's turned on, then the image will rotate. The second row shows the current temperature and in the third row the user is able to set/remove the temperature condition.

If the fan is on, the image rotates. This feature is implemented with JavaScript and for that to happen also the library from <http://raphaeljs.com> is used.



---

**Algorithm 45** Java EE Server - Fan JavaScript

---

```
var angle = 0, image, interval;

//Initializes the fan with help from the library 'Raphael - JavaScript
  Library'
function initFan(divId, imgSrc) {
    var R = Raphael(divId, 100, 100);
    image = R.image(imgSrc, 0, 0, 100, 100);
}

//Sets an interval with 7ms, which turns the image by 5 degrees to the
  right
function startFan() {
    clearInterval(interval);
    interval = setInterval(function() { image.animate({ transform: "r"
        + angle }, 1, ""); angle += 5; }, 7);
}

//Clears the interval and therefore stops the rotating fan
function stopFan() {
    clearInterval(interval);
}
```

---

In the code above you are able to see that the image of the fan is setted in the 'init-Fan(divId, imgSrc)' with the help of the library seen in 45. In the 'startFan()' function the code creates an interval and every interval poll the image is rotated by 5 degrees. Here again the mentioned library is used. The 'stopFan()' function clears the interval and therefore stops the fan.

---

**Algorithm 46** Java EE Server - Webinterface Fan

---

```
...
//Initializes the fan with the div below and the picture
<script type="text/javascript">
    initFan('rotateDiv#{deviceController.selectedDevice.deviceId}', 'pictures
        /fan.png')
</script>

//If the state of the fan is on, the fan starts
<c:if test="#{deviceController.currentFan.state eq 'on'}">
    <script type="text/javascript">
        startFan();
    </script>
</c:if>

//Stops the fan if the state is off
<c:if test="#{deviceController.currentFan.state eq 'off'}">
    <script type="text/javascript">
        stopFan();
    </script>
</c:if>

...

//The div of the fan
<div id="rotateDiv#{deviceController.selectedDevice.deviceId}" />

//According to the status of the fan, the right button is displayed
<c:choose>
    <c:when test="#{deviceController.currentFan.state eq 'off'}">
        <p:commandButton value="Turn On"
            action="#{deviceController.currentFanTurnOn()}"
            onclick="startFan();"
            update="@form" />
    </c:when>
    <c:when test="#{deviceController.currentFan.state eq 'on'}">
        <p:commandButton value="Turn Off"
            action="#{deviceController.currentFanTurnOff()}"
            onclick="stopFan();"
            update="@form" />
    </c:when>
</c:choose>

...

```

---

In the upper part of this code the fan is initialized and will be started if the fan is turned on. In the lower part the fan can be started or stopped with the buttons and according to the state the right button is displayed.

All other devices:

- Heater
- AlarmSystem
- PowerOutlet
- Light

have a quite simple user interface. They are also built up with a grid and have buttons and pictures attached to it.

## Profiles

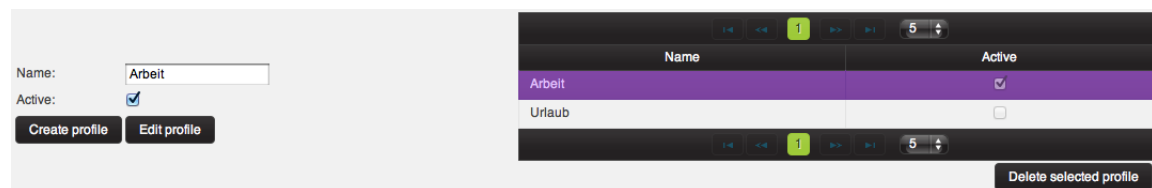


Figure 45: Webinterface Profile

As you can see in the picture 45 all profiles are shown in the table. The user is able to create, edit and delete these profile by selecting the profile in the table or simply write a name for a profile in the textfield and create it.

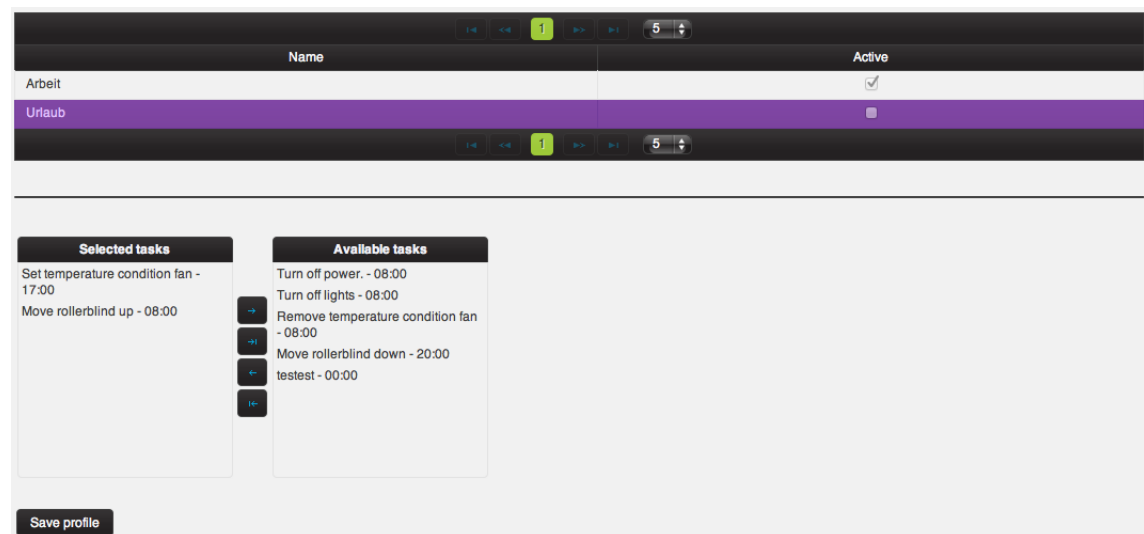


Figure 46: Webinterface add Tasks to Profile

In the picture 46 the user is able to add the tasks to the profile. The user is able to select an existing profile and therefore he is able to see the tasks which are already included in

the profile and the tasks. Also all other tasks are shown and can be included or removed from the profile.

## Settings

Place	IP - Address	Logging activated	
<input type="text" value="Local"/>	<input type="text" value="192.168.1.2"/>	<input checked="" type="checkbox"/>	<a href="#">Delete place</a>
<input type="text" value="Home"/>	<input type="text" value="215.125.25.76"/>	<input checked="" type="checkbox"/>	<a href="#">Delete place</a>

[Add place](#)

Figure 47: Webinterface Settings

In the settings tab the user sees all MCPPlaces. He is able to create/edit/delete them. When the checkbox for logging is selected or deselected the server will instantly send the accordingly CallMessage to the arduino of the specified IP - Address. An important point is also that the 'Retrieve devices' - button is in this tab. If the user press this button a appropriate CallMessage is sent to all MCPPlaces listed.

### 3.3.9 Logging Arduino

The system has to run 24 hours a day, 7 days a week and therefore a logging system is needed to ensure the stability. Logs are stored and transmitted in LogMessages.

Any Control App, Actor or Sensor which wants to make use of logging, only needs to call the static Log classes method named log. It accepts two parameters: the Logmessage and the logLevel which is defined as follows:

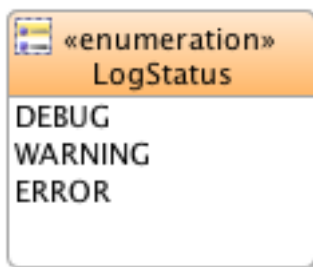


Figure 48: LogStatus

All preferences related to the logging process are implemented in the `SystemLogPreferences` class. It can be set to which destinations logging should be done and also to which level debugging should be performed. This class holds a vector of `Connection` types which are described in 50. Connections can be added and deleted from this vector. The level of debugging can for example be set to `ERROR` so that only `ERROR` messages are sent. Another possible configuration would be to set it to `DEBUG` and in further consequence all messages will be sent.

The following steps must be done to construct and to invoke an example `LogMessage`:

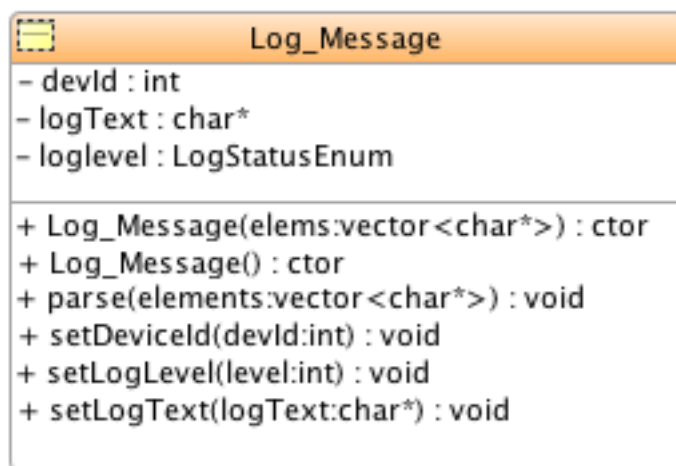


Figure 49: Log\_Message Class

**Algorithm 47** Arduino Create Log\_Message

---

```

//generating a new instance of the Log_Message class
Log_Message *msg = new Log_Message();
//setting the deviceId
msg->setDeviceId(this->getId());
//allocate the character Array
char *text = (char*) malloc(30);
//fill the character Array with text
sprintf(text, "%s_was_turned_on", this->getName());
//setting the text
msg->setLogText(text);
//sending the logMessage with logLevel=verbose
Log::verbose(msg);
delete(msg);

```

---

In the first step, the message is allocated. Then the deviceId is set properly and the charArray named text is allocated, filled and set. The last step sends the Logging message to the destinations configured in the SystemLogPreferences class.

**3.3.10 Connection Handling**

Basically, all connections are abstracted into one layer, one class named Connector. In this class, connections are established and closed and data are also sent and received.

The principle of the Connection class is, that when a message is received, a callback method is being invoked. The different connection types, as there are ADK connections, serial connections and ethernet connections over socket are all handled individually, but the result is the invocation of the callback method. Every connection type is periodically polled and it is proved if there is data available.

Every ControlApp has the ability to get the last connection using the Connector class. Connections can then be sent back to the stored connection using a Connection object as argument. The Connection struct is defined as follows:

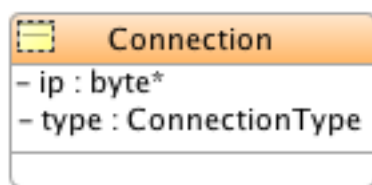


Figure 50: Connection Struct

It can be retrieved using the `getLastConnection` Method from the `Connector` struct. The struct contains an IP - address and a connection type, this is an enum and is defined as follows:

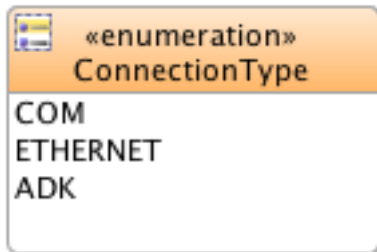


Figure 51: ConnectionType

### Serial Connections Initialization

To initialise serial connections, `initSerial` has to be called which is defined as following:

---

**Algorithm 48** Arduino `initSerial`

---

```
void Connector::initSerial()
{
    Serial.begin(BAUDRATE);
}
```

---

It is a pretty straightforward method calling only `Serial.begin` which is a function from the Arduino framework from now on allowing to send and receive data through the serial interface.

### Receiving Messages

Periodically, `checkForSerialMessages` is called. It proves if there is a message available and then, when received invokes the callback. If the connection is established and available, the following code is invoked:

---

**Algorithm 49** Arduino Serial receive

---

```
delay(5); //waiting for new charakters to arrive
//reading one charakter
c = (char) Serial.read();
//adding the end of the string
int len = strlen(mcpMessage);
mcpMessage[len] = c;
mcpMessage[len+1] = '\0';
```

---

The little delay here is needed, otherwise messages will not be fully transmitted. with the Serial.read method, one character of the message is received and then it is added to the charArray. Due to the fact that strings in C normally end with an \0 as last character, it is added dynamically.

Once it is fully received, the callback method is called using the following line of code:

---

**Algorithm 50** Arduino Connector Callback

---

```
(listener->*messageCallback)(mcpMessage);
```

---

## **Sending Messages**

Sending messages is quite simple using the defined Arduino framework function named Serial.println() or Serial.print.

## **Ethernet Connections**

### **Initialization**

To initialize ethernet connections, initEthernetShield has to be called which is defined as following:

---

**Algorithm 51** Arduino Ethernet Initialization

---

```
//defining the mac address of the arduino
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
//defining the ip address of the arduino
byte ip[] = { 192,168,1, 2 };
void Connector::initEthernetShield()
{
    //allocating the destination ip adress
    this->current.ip= (byte*) malloc(sizeof(uint8_t)*4);

    Ethernet.begin(mac, ip);
    server.begin();
}
```

---



In the first step, the Connector's class variable of the type Connection is newly allocated for further use. Then Ethernet is enabled using the Arduino framework functions named Ethernet.begin and server.begin. Ethernet.begin enables the ethernet shield and makes it available for further use using the mac address and the IP-Address as parameters. Server.begin enables a tcp server on port 13316 for incoming connections.

### Retrieving Messages

Receiving messages over a socket connections is basically the same as retrieving strings over the serial Interface once the connection is established. Establishing connections is made as follows:

---

**Algorithm 52** Arduino Retrieving Ethernet Messages

---

```
EthernetClient client = server.available();  
if (!client) return;  
while(client.available)  
{  
    char c = client.read();  
    //input processing  
}
```

---

At first, we had to prove whether a client is available. Then, while data is available, input processing is done periodically.

### Sending Messages

Messages are sent through the Ethernet interface using nearly the same way as retrieving. At first the connection is newly established using

---

**Algorithm 53** Arduino Client Connect

---

```
client.connect(<ip-Adress>,13316);
```

---

then messages can be easily send through the open socket connection using

---

**Algorithm 54** Arduino Client Send Message

---

```
client.println(message);
```

---

Afterwards the socket is flushed and closed.

## ADK Connections

### Preparation

To make use of the ADK shield, two Arduino libraries must be included in the Arduino's libraries folder. They are officially published by Google and can be found at: <http://developer.android.com/guide/topics/usb/adk.html>. The first library named USBHostShield is used for communicating to any device plugged in. You could, for example plug in an external mass storage device and write your own drivers for that. The second library is the USBAccessory library which mainly handles the communication with the Android based opponent.

### Initialization

First of all, on the top of the file, the USBAccessory library and the USBHostShield library have to be included. Then the following initialization code has to be inserted:

---

#### Algorithm 55 Arduino ADK Initialization

---

```
AndroidAccessory acc("<ProducerName>", "<ApplicationName>", "<Description>"  
    , "<Version>", "<link>", "<serial_number>");
```

---

### Retrieving Messages

Retrieving Messages is then as easy as retrieving serial messages.

---

#### Algorithm 56 Arduino ADK retrieving Messages

---

```
int len = acc.read(buffer, sizeof(buffer), 1);
```

---

### Sending Messages

Sending Messages is done with only one line of code:

---

#### Algorithm 57 Arduino ADK sending Messages

---

```
acc.write(erg, strlen(erg));
```

---

There is no `writeln` Method so if a whole line is sent, as it is in our case, a `'\n'` character has to be added at the back.

### 3.3.11 Event Handling

Every `ControlApp` has the ability to send messages at any time to the client, but it has to know to which client the message will be sent. Normally, events are established with an asynchronous sending of `Call-` and `ResultMessages` over a period of time.

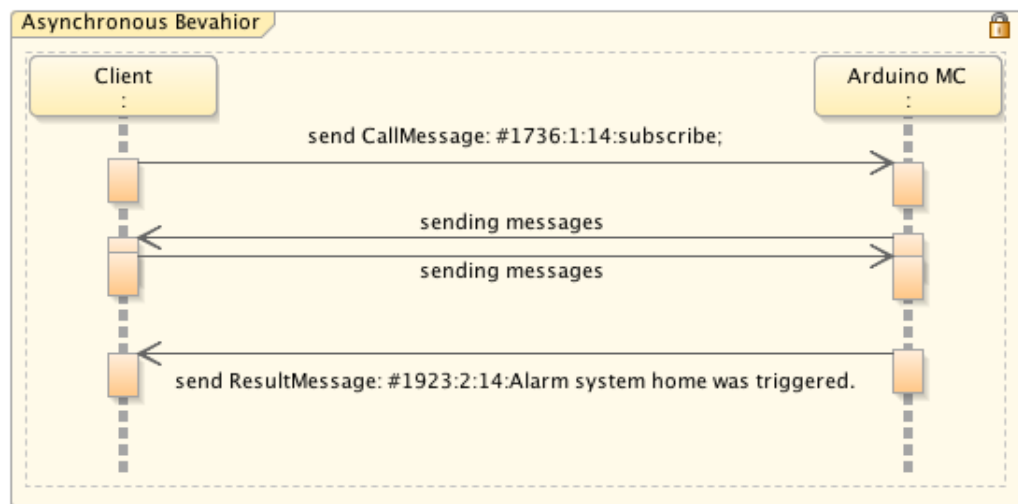


Figure 52: Asynchronous Behavior

When the client is subscribing for the specific event, there is no need for sending the IP - address as a parameter. Instead, every Control App can get the current connection struct by simply calling

---

**Algorithm 58** Arduino GetLastConnection
 

---

```
Connector::getInstance()->getLastConnection();
```

---

Then the Control App itself stores the connection and when the event is called, the ResultMessage has to be constructed and send back by calling

---

**Algorithm 59** Arduino SendResult
 

---

```
Connection::getInstance()->sendResult(msg, connection);
```

---

With this mechanism, events can be created using subscribe methods dynamically.

### 3.3.12 Android Application

In course of developing this diploma thesis, a sample app for Android based tablets was implemented, which is developed to show the possibilities that our system offers. The app was implemented using Google API version 3.1. It allows the steering of a rollerblind ControlApp, meaning that the rollerblind can be moved, the condition can be set and

the environment variables can be seen in the main interface using a well designed user interface. Implementing the protocol was quite easy, due the fact that the protocol is also implemented in Java, so only minor changes had to be made. Interpreting the messages on the other hand, was completely new implemented.

When a message is recieved, whether through a socket connection or directly via the ADK, the `messageInterpreter` class is called wich parses the recieved line. The `interpret` method is partly inherited from the Java EE server, but some aspect have to be changed.

---

**Algorithm 60** Android Device Reflection

---

```
//retaining the class with the packagename
Class<?> cls = Class.forName("entity.device." + deviceMessage.getDeviceType
    ());
//retrieving the appropriate konstrutor
Constructor<?> c = cls.getConstructor(new Class[] { int.class, String.class
    , String.class, String.class, String.class });
//little adaptions for use in the Arduino system
deviceMessage.setCurrentState(deviceMessage.getCurrentState().substring(0,
    deviceMessage.getCurrentState().length() - 1));
//creating a new instance of this device
device = (Device) c.newInstance(deviceMessage.getId(), deviceMessage.
    getName(), connection, deviceMessage.getDeviceType(), deviceMessage.
    getCurrentState());
//add this device to the Devicemanager's list
DeviceManager.getInstance().add(device);
```

---

The reflection code seen above is needed because in the Arduino system is not working with databases. Instead, Devices are dynamically instanciated using reflection and then added to the global singleton class named `DeviceManager`. The `DeviceManager` class offers methods to add listeners when a new device is added. So when `DeviceManager.getInstance().add(device)` is called, the listener is also informed.

On the GUI side, the `DeviceAddedListener` is implemented wich is defined as follows:

---

**Algorithm 61** Android DeviceAddedListener

---

```
public interface DeviceAddedListener {
    public void onDeviceAdded(Device d);
}
```

---

In the main activity this interface is implemented and the relevant devices are filtered and stored. Results wich are sendet after every call made, can also be retrieved using the following interface.

---

**Algorithm 62** Android ResultMessageListener

---

```
public interface ResultMessageListener {  
    public void onResultMessageRecieved(ResultMessage msg);  
}
```

---

The main activity implements both interfaces and adds itself as listeners using the following code:

---

**Algorithm 63** Android Listener Subscribing

---

```
DeviceManager.getInstance().addListener(this);  
MessageInterpreter.getInstance().addResultMessageListener(this);
```

---

Once the connection is established using the permission code described in 3.1.7, a new Thread is started which polls the SensorDevice to get the Brightness and the Temperature every two seconds.

---

**Algorithm 64** Android SensorDevice Thread

---

```
Thread t = new Thread(){  
    @Override  
    public void run(){  
        while(true)  
        {  
            try  
            {  
                //wait before polling  
                sleep(2000);  
            } catch (InterruptedException e)  
            {  
                System.err.println(e.toString());  
            }  
            if (mSensorDevice!=null) {  
                //send a CallMessage  
                mSensorDevice.brightnessRCS();  
                //send a CallMessage  
                mSensorDevice.temperatureRCS();  
            }  
        }  
    }  
};  
t.start();
```

---

When Resultmessages are received, the listener method named OnResultMessageReceived gets called. When the deviceId equals the sensorDevice's id, the result is interpreted and showed on the UI.

---

**Algorithm 65** Android ResultMessage Receiver

---

```
mTextViewTemperature.post(new Runnable() {  
    public void run() {  
        if(msg.getDeviceId() == mSensorDevice.getDeviceId()){  
            if(msg.getResult().split("=")[0].equals("temperature")){  
                //setting the current Temperature textView  
                mTextViewTemperature.setText(erg.split("=")[1]+ "°C");  
            }  
        }  
    }  
});
```

---

Direct changing of the UI from a thread is dangerous and mustn't be done. So the post method is used, where the changing is posted into the UI Queue and then executed using the Runnable interface.

Therefore, a listener thread has to be started which continuously checks whether new Messages are available.

**Algorithm 66** Android Retrieving ADK Messages

---

```
while(true)
{
    try{
        //wait for new messages to arrive
        Thread.sleep(100);
        //create a buffer for retrieving
        byte[] buffer = new byte[16384];
        int ret = mInputStream.read(buffer);
        line = "";
        for(int i=0;i<ret;i++) {
            //appending the charakter to the string line
            line += (char)buffer[i];
        }
        if(line!=null) {
            //successfully received
            mInterpreter.interprete(line, "ADK");
        }
    }
    catch (IOException e) {
        System.err.println(e.toString());
    }
    catch (InterruptedException e) {
        System.err.println(e.toString());
    }
}
```

---

Before this step, the system checks whether ADK connection is available and if not proves if a socket connection is available. Opening a `BufferedReader` on this `InputStream` will result in receiving no data. So the data is readed charakter for charakter. When the message is received, the interpreter is called using „ADK“ as connection parameter.

**UI** The UI design is pretty simple, offering as much usability as possible.



Figure 53: Android User Interface

On the left side, a rollerblind is showed, offering the ability through the vertical slider on the right side to move the rollerblind up and down in realtime. Once the slider is released, a callmessage is sent to the rollerblind device with the current position as parameter. On the right side, the current brightness and the current temperature is shown, changing their values every two seconds. Below this area, the condition can be changed. It describes on wich brightness the rollerblind should move up. If the brightness falls below this point, the rollerblind moves up again.



## 4 Selfevaluation

### 4.1 Personal experience

#### 4.1.1 Java EE

##### Primefaces & UI

Overall programming in Java EE is quite nice. I had several problems, but the main one was in building the ui. I wasn't able to figure out how exactly the property 'update' in the primefaces elements works. I surely spent about 30 hours on this single problem. In the end I had to look in the sourcecode of the browser to get the correct id from the element and copy it into the update property of the appropriate element.

##### Glassfish

Another problem I experienced was with the application server Glassfish. Sometimes it was impossible for me to know or to read from the exception from where an error occurred. Because of these exceptions I had to create a new project, copy the whole code from my old project into the newly created one and then work in the new one onward. I did this about ten times and it worked again afterwards with the exactly same code.

#### 4.1.2 Arduino

**Working with Strings** Due to the fact that the Arduino framework offers the ability to work with Strings, we made our life easier by making use of this library. But in fact, we had found one problem which is a critical issue and should be fixed as fast as possible. When adding String objects with the „+“ operator, every now and then, the pointer is not calculated successfully and so the output was outside the Strings scope. When invoking this issue about ten times or more, the Arduino system restarts without any other exception message.

---

**Algorithm 67** Arduino Working with Strings

---

```
const String args[3] = { "foo", "bar", "baz" };
while(true)
{
    String result = "";
    Serial.println(result);
    result += args[0];
    Serial.println(result);
    result += args[1];
    Serial.println(result);
    result += args[2];
    Serial.println(result);
    Serial.println();
}
```

---

The code above demonstrated this issue and shows that if compiled and started, the Arduino system will restart within seconds over and over again. Further discussion for this bug can be found at [5]

**Getting the remote Ip-Address** The Ethernet shield, is equipped with the Wiznet W5100 chip which is rather powerful. The Arduino library named Ethernet is basically a layer around this chip for easy using. Due to this fact, it happens that not all functions available on the chip are packed in the layer. In our case we wanted to know from which IP-Address the connection is established, but there was no function available offering this availability. We discovered in the datasheet, that the ethernet chip has the ability to read this address using custom registers defined in the datasheet. These four registers are named Sn\_DIPR and here is our code for reading these registers:

---

**Algorithm 68** Arduino Getting the Remote Address

---

```
void W5100Class::getSn_DIPR(uint8_t * addr)
{
    addr[0] = read(0x040C); //192
    addr[1] = read(0x040D); //168
    addr[2] = read(0x040E); //0
    addr[3] = read(0x040F); //2
}
```

---

**Using std::map** The std::map template which we used pretty often, has a bug when accessing elements. Now and then, when the key is set to int and we tried to access this element, null is returned. But when iterating through the map, and comparing each key with another key, the result is returned properly. So instead of calling:

---

**Algorithm 69** Arduino Int Map

---

```
return map[index];
```

---

We made a simple workaround:

---

**Algorithm 70** Arduino Map Problem

---

```
std::map<int, Actor*>::iterator itx;
for ( itx=actorMap.begin(); itx != actorMap.end(); itx++ )
{
    //prove if the key = the searched id
    if(id==itx->first) return itx->second;
}
```

---

**Linker Problem** When a library includes a file and the main sketch does not include this file, the linker will not find the file and the whole compiling process will fail. So there is only one option, to include all relevant files in the main sketch.

**Memory Problem** When implementing our system on the Arduino platform, we stretched the Arduino system to it's limits. We could never run all ControlApps during the testing phase at the same time. Due to the limited SRAM resource of 8 kBytes, every step made was designed to be memory friendly, but still, we weren't able to integrate all apps.

We used many techniques for creating the process more dynamically, which we wanted to archive, but we didn't thought of having such problems concerning memory use. These techniques were mainly developed for much larger system and more SRAM. SRAM can be extended using external chips, but within this prozess, the Arduino has to be resoldered.

## **4.2 Final words**

All in all working with these technologies was quite impressing and very complex. It opens up new perspectives to know how to connect hardware with software in a efficient way. While programming this diploma thesis we got deep insights into C++, which we never really learned in our lessons, and very specific Java problems. We are very happy that we were allowed to work on this diploma thesis and therefore in this section.

Special thanks to our professors, Thomas Stütz and Gerald Köck, who supervised and supported us whenever we had problems concerning this thesis.

## 5 Source directory

- [1] Arduino sdfat library. <http://code.google.com/p/sdfatlib/>.
- [2] C++ standart library. <http://www.cplusplus.com/reference/>.
- [3] Standart template library - maps. <http://www.cplusplus.com/reference/stl/map/>.
- [4] Java definition, 10 2000. <http://searchsoa.techtarget.com/definition/Java>.
- [5] Arduino: Difficulty with string concatenation. forum entry, 04 2011. <http://stackoverflow.com/questions/5782772/arduino-difficulty-with-string-concatenation>.
- [6] Android open accessory development kit, 5 2012. <http://developer.android.com/guide/topics/usb/adk.html>.
- [7] Jpa, 05 2012. <http://de.wikipedia.org/wiki/JPA>.
- [8] Antonio Goncalves. Beginning java ee 6 platform with glassfish 3. Apress, 2010. Second Edition.
- [9] Daniel Shiffman. Interview with casey reas and ben fry. <http://rhizome.org/editorial/2009/sep/23/interview-with-casey-reas-and-ben-fry/>.
- [10] Phillip Torrone. Why google choosing arduino matters and is this the end of "made for ipod" (tm)? <http://blog.makezine.com/2011/05/12/why-google-choosing-arduino-matters-and-the-end-of-made-for-ipod-tm/>.