

2013

Voet Bart

2014

Monitoring temperatuur en relatieve vochtigheid voor onderzoek naar energieuwige woningen

Ondernemingsproject voorgedragen tot het behalen van het diploma
van Gegradueerde in de Elektronica

Promotor:

De heer Gert Appels

Copromotor:

De heer Ralf Klein

KU Leuven - Departement bouwkunde

2013

Voet Bart

2014

***Monitoring temperatuur en relatieve vochtigheid
voor onderzoek naar energieuinige woningen***

Ondernemingsproject voorgedragen tot het behalen van het diploma
van Gegradeerde in de Elektronica

Promotor:

De heer Gert Appels

Copromotor:

De heer Ralf Klein

KU Leuven - Departement bouwkunde

Table of Contents

PART 1: INTRODUCTION.....	1
1 PREFACE.....	1
1.1 PROBLEM-STATEMENT.....	1
1.2 COLLABORATION.....	1
1.3 REQUIRED FUNCTIONALITY IN A NUTSHELL.....	1
1.4 POSITIONING.....	2
2 CONTEXT-SETTING.....	3
2.1 CONTEXT.....	3
2.2 HVAC-SYSTEM.....	4
2.3 STAKEHOLDERS.....	5
2.3.1 <i>Division of Building Physics KUL (Department Civil Engineering)</i>	5
2.3.2 <i>ACE Group T</i>	5
2.4 SCOPE.....	6
2.4.1 <i>Digital sensors</i>	7
2.4.2 <i>Integration</i>	7
3 STRUCTURE.....	8
3.1 PART 1 : INTRODUCTION.....	8
3.2 PART 2: FIELD STUDY.....	8
3.3 PART 3: REQUIREMENTS.....	8
3.4 PART 4: BUILDING-BLOCKS AND ARCHITECTURE.....	8
3.5 PART 5: ROLL OUT.....	8
3.6 PART 5: CONCLUSIONS.....	8
4 APPROACH.....	9
4.1 INCEPTION (PART 1).....	10
4.2 PRE-STUDY (PART 2).....	10
4.3 REQUIREMENT ANALYSIS (PART 3).....	10
4.4 RESEARCH ON WHAT WE CAN REUSE (PART 3-4).....	10
4.5 ARCHITECTURAL DESIGN (PART 4).....	11
4.6 BUILDING-BLOCKS (PART 4).....	11
4.6.1 <i>Sensors</i>	13
4.6.2 <i>Sensor-connectivity</i>	13
4.6.3 <i>Devices and hardware</i>	13
4.6.4 <i>Software</i>	14
4.6.4.1 <i>Driver</i>	14
4.6.4.2 <i>Libraries</i>	14
4.6.4.3 <i>Application</i>	14
4.6.4.4 <i>Tools and frameworks</i>	14
4.6.4.5 <i>Storage</i>	14
4.6.4.6 <i>Integration</i>	15
4.6.5 <i>Configuration and documentation</i>	15
4.7 BUILD AND ASSEMBLE (PART 5).....	16
PART 2 : FIELD-STUDY.....	17

5 SENSORS (FIELD-STUDY).....	17
5.1 INTRODUCTION AND STRUCTURE.....	17
5.2 WHAT IS A SENSOR?.....	17
5.3 SENSOR-CHARACTERISTICS.....	18
5.3.1 Range(s).....	19
5.3.2 Linearity.....	19
5.3.3 Electrical specifications.....	20
5.3.4 Speed and performance.....	20
5.3.5 Accuracy.....	21
5.3.6 Precision.....	21
5.3.7 Resolution.....	21
5.3.8 Drift.....	22
5.3.9 Long term drift.....	22
5.3.10 Environmental conditions.....	22
5.3.11 Transfer functions.....	23
5.3.12 Heat and power-dissipation.....	23
5.3.13 Sensitivity.....	23
5.3.14 Compatibility.....	23
5.3.15 Interfering Inputs vs modifying inputs.....	24
5.3.16 Hysteresis.....	25
5.3.17 Error.....	25
5.4 EVOLUTION OF SENSORS.....	26
5.4.1 Smart sensors.....	26
5.4.2 Connection-protocols (for smart sensors).....	27
5.4.2.1 I2C.....	27
5.4.2.2 SPI.....	28
5.4.2.3 Digital Resistance (ON/OFF).....	29
5.4.2.4 Serial Port (UART).....	29
5.4.2.5 Bit-Banging.....	29
5.4.3 Sensor-networks.....	30
5.4.3.1 Sensor-node.....	30
5.4.3.2 Data node.....	30
5.4.3.3 Aggregation-node.....	30
5.4.4 Sensor-networks in the context of this project.....	31
5.5 SENSORS IN THE DOMAIN OF BUILDING PHYSICS.....	32
5.5.1 Relative humidity.....	32
5.5.2 Temperature.....	32
5.5.3 Differential pressure.....	32
PART 3 : REQUIREMENTS.....	33
6 REQUIREMENTS AND CONCERNS OF THE SYSTEM.....	33
6.1 INTRODUCTION AND STRUCTURE.....	33
6.2 SYSTEM USAGE (SCENARIO'S AND ACTORS).....	34
6.2.1 Actors/roles.....	34
6.2.2 Education and classrooms.....	34
6.2.3 Specialized building accommodation for academic research.....	35
6.2.4 Other academic research on buildings.....	35
6.3 FUNCTIONAL REQUIREMENTS.....	36
6.3.1 Controlling sensors at runtime.....	36
6.3.2 Collect data from sensors.....	36
6.3.3 Correlation of measurements to configuration.....	36
6.4 CONCERNS AND QUALITY ATTRIBUTES OF THE SYSTEM.....	37
6.4.1 Operability, monitoring.....	38
6.4.2 Maintainability and modifiability.....	39

6.4.3 Configurability and flexibility.....	40
6.4.4 Usability.....	41
6.4.5 Hierachy/Modularity/Regularity/Encapsulation.....	42
6.4.6 Reliability, availability, durability, autonomy, resilience.....	43
6.4.7 Scalability, mobility, portability.....	44
6.4.8 Scalability and performance.....	45
6.4.9 Identification, correlation and timing.....	46
6.4.10 Interoperability.....	47
6.4.11 Testability.....	47
6.4.12 Security.....	47
6.4.13 Open standards and open source.....	48
6.4.14 Accessibility.....	49
PART 4: ARCHITECTURE AND BUILDING BLOCKS.....	50
7 ARCHITECTURAL DESIGN.....	50
7.1 SUMMARY OF REQUIRED SYSTEM-CAPABILITIES.....	51
7.2 LAYERING OF THE SYSTEM.....	51
7.3 SENSOR-LIBRARY (AND API).....	52
7.4 SENSOR-AGENT.....	53
7.5 SENSOR-HUB.....	54
7.6 INTEGRATION AND NETWORK.....	55
7.6.1 Messaging.....	55
7.7 MONITORING.....	56
7.8 REFERENCE IMPLEMENTATION AND SPECIFICATIONS.....	57
8 ELABORATION OF BUILDING BLOCKS.....	58
8.1 TECHNOLOGY-STACK: LANGUAGES (AND PLATFORMS): C AND JAVA.....	59
8.1.1 C-libraries.....	59
8.1.2 Java.....	60
8.2 PRINCIPLE APPLIED: MODULARITY AND DEPENDENCY-INJECTION.....	61
8.2.1 Definition of modularity.....	61
8.2.2 Dependency injection.....	61
8.2.3 Applying modularity towards platforms.....	61
8.2.3.1 Java.....	61
8.2.3.2 C.....	61
8.3 MODULE: SENSOR-API (LIBRARY).....	62
8.3.1 Context.....	62
8.3.2 Library with API.....	62
8.3.3 Composition of the library.....	63
8.3.4 System-abstraction.....	64
8.4 MODULE: SENSOR-AGENT.....	65
8.4.1 Sensor-agent is a process.....	65
8.4.2 Starting up the SensorAgent.....	66
8.4.3 Measurement-process.....	67
8.4.4 (Re)Configuring the Sensor.....	68
8.4.5 Local storage.....	68
8.4.6 Integrating and events.....	69
8.4.7 Implementations for SensorRegistry.....	70
8.4.8 Implementations for SensorConfiguration.....	70
8.5 COMPONENT: SENSOR-HUB.....	71
8.5.1 Composition of the Sensor-hub.....	71
8.5.2 Event messages.....	72
8.5.2.1 New configuration pushed to the sensor-agent.....	72

8.5.2.2 Measurement pushed via the sensor-agent.....	73
8.5.2.3 Activation of sensor and sensor-agents.....	73
8.6 DATA-STORAGE.....	74
8.6.1 <i>Local and remote storage</i>	74
8.6.2 <i>Data-model for Sensors</i>	74
PART 5: ROLL OUT.....	75
9 GIT AND GITHUB.....	75
9.1 DOCUMENTATION AND CODE FOR THIS PROJECT.....	76
10 SUPPORTING MODULARITY IN JAVA.....	77
11 SUPPORTING MODULARITY IN C.....	80
12 PROCESSING DEVICES.....	81
12.1 SET UP WITH RASPBERRY PI.....	81
12.2 ARDUINO MICRO PRO 3,3 v.....	82
13 INTERACTING WITH THE APPLICATION (REST-FULL SERVICES)....	83
13.1 WHAT IS REST?.....	83
13.2 WHAT IS JSON.....	83
13.3 LIST OF AVAILABLE REST-FULL SERVICES.....	84
PART 6 : CONCLUSION.....	85
14 CONCLUSION.....	85
14.1 SENSOR-SUPPORT.....	86
14.2 SUPPORT FOR MULTIPLE DEVICES.....	86
14.3 NETWORK-INTEGRATION.....	86
14.4 CLIENT-INTEGRATION.....	86
15 APPENDICES.....	87
16 BIBLIOGRAPHY.....	88

List of Figures

List of Tables

Appendices

Appendix I: Heading of this appendix.....	78
---	----

List of Abbreviations and Symbols

BT	Bachelor thesis
KUL	Katholieke Universiteit Leuven
ACE Groep T	Anticipative Continuing Education Groep T

PART 1: Introduction

1 Preface

1.1 Problem-statement

The Department of Building Physics of the University of Leuven is doing research on energy efficient buildings.

For this, they have a great demand for reliable and inexpensive system(s) for continuous measurement of parameters that are important for indoor climate and energy consumption.

1.2 Collaboration

To tackle these requirements a collaboration has been set up between the University of Leuven and ACE Groep T.

This thesis - as a result of this collaboration - describes the research, design and implementation of a system that should allow the concerned actors (researchers and students) to interact with various kind of sensing-devices.

1.3 Required functionality in a nutshell

Very briefly the functionality that is expected from this **system**:

- It allows the users to interrogate and control one or more sensors concurrently
- It can collect data from the sensors and provisions data for further processing
- Connects to different types of sensors via various protocols (i2c, spi, custom, ...)
- It allows users to configure various parameters of the sensors

In addition to these functionalities the **system** should be

- Extensible (plug and play) for new types of sensors
- Built with open technology and does not enforce vendor locking
- Easy to set up and maintain
- Very well documented

1.4 Positioning

Basic goals of this thesis is to deliver a way of configuring, interacting and extracting data from a variety of sensors (e.g. temperature, relative humidity, pressure, ...).

The primary stakeholders are to be able to integrate this result in their existing research in the field of building physics.

This thesis has **not** the intention to do **academic** research on sensors itself, its primary goal is to **investigate** the **integration** of sensors into the research towards energy-efficient buildings.

In short this thesis is the result (and output) of the following activities :

- Investigation towards the needs (functional and non-functional) of this sensor-integration.
- Evaluation and comparison of existing techniques used for this integration.
- Development, configuration and deployment of this integration on both hardware as software-level.
- Structured documentation of this integration, sensors and other components (hardware or software)

2 Context-setting

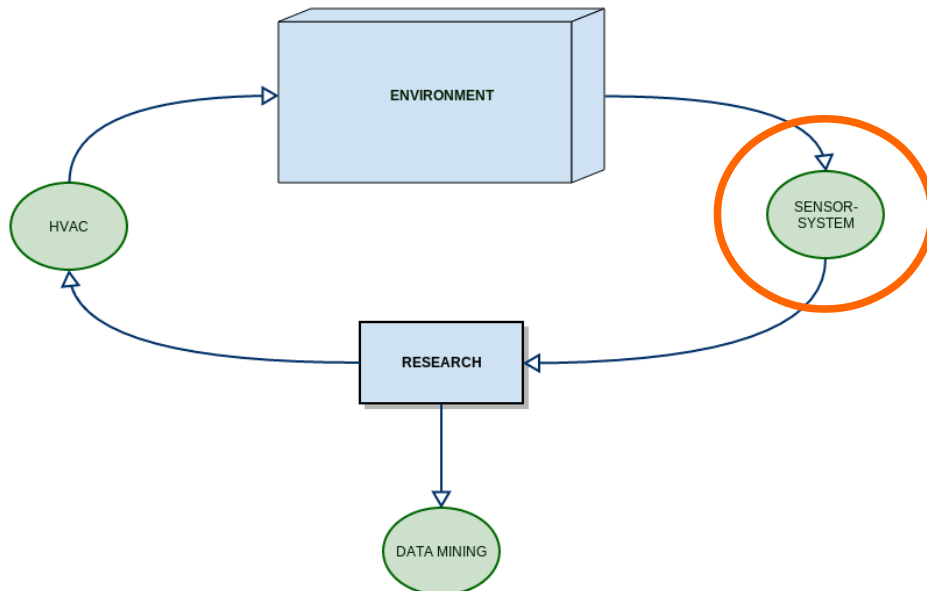
This chapter describes - before diving into the actual content - the research-process from a 10,000 feet-view.

It also identifies the stakeholders and describes their specific interests and needs related to this thesis.

The last part of this chapter concludes with a scope-demarkation and boundary-definition on the subject of this thesis.

2.1 Context

The current research (this thesis needs to support) is an investigation on how specific building constructions react on changes of specific environment parameters.



The **main research** (and use case for this thesis) is applied in a **controlled** environment and consists of the following activities:

- Researchers manipulate and control different parameters (e.g. temperature, ventilation, ...) via a HVAC-infrastructure to a building-infrastructure to test and measure the effect these changes.
- These changes need to be measured and to be kept in a data-store
- The measurement-data is to be correlated with the (sequences of) manipulations
- This correlated data is investigated

Beside the research within this HVAC-infrastructure of domains of research are to be supported in less controlled environments like churches, public buildings (with less or no infrastructure).

2.2 HVAC-system

The HVAC-infrastructure deployed in this research-facility allows the researchers to manipulate the environment and to run/trigger specific sequences and algorithms.

HVAC (heating, ventilation, and air conditioning) is the technology of indoor and vehicular environmental comfort.



If you want to learn more about this research-facility and its infrastructure you can read the document added to the appendix of this thesis about the lecture rooms of the KAHO.

2.3 Stakeholders

2.3.1 *Division of Building Physics KUL (Department Civil Engineering)*

As already indicated by the context, the **first** requirement for the KUL (primary stakeholder) is the design, development and documentation of a system that **captures data** from sensors within the research-facility.

This system should be **stable** enough to support continuous monitoring and data-capturing (for a long periods) and should be stable enough to **recover** from infrastructure problems like blackouts of the electricity or network.

The system should also **scale** to an **educational** environment.

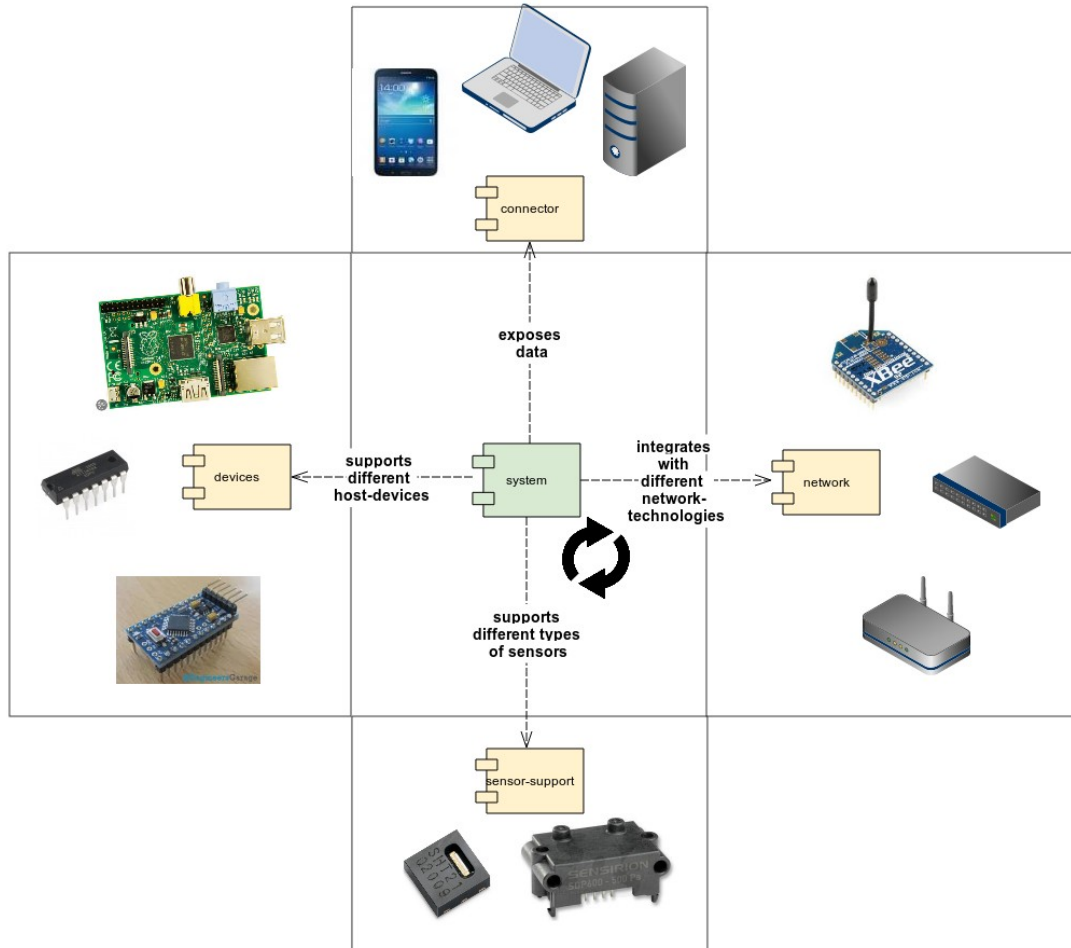
Students of the KUL should be able to use this system to start learn working with these sensors (in the context of classical exercises or homework).

2.3.2 *ACE Group T*

Although not the main objective of this thesis, the focus on sensors with this thesis can contribute to the content of the current material surrounding the topic of embedded development .

2.4 Scope

This thesis focusses on the design and development of a **system** that enables different actors to control and collect data from sensors.



The core idea behind this thesis is to set up and design a system that is pluggable and scalable towards different scenario's:

- It should be able to run on different devices depending on the scenario's
- Different types of sensors should be supported
- The design is open enough to integrate in different network-technologies (Ethernet, WIFI, RF, Zigbee, Meshing,)
- It exposes data via an protocol open enough to support different types of clients (other servers, monitoring-systems and

but ...

2.4.1 Digital sensors

The type of sensors we focus on for building this system are modern sensors which expose their measurements via a digital interface like i2c, SPI or UART, ...

The thesis isn't about developing new sensors or research linked to do that. It's not a request of the stakeholders (a selection of sensors had been made at the inception of the project).

It focusses on building out a framework for facilitate the interaction with modern sensors. This doesn't imply that “classic sensors” are not supported by framework, the software described in the thesis should be flexible enough to plug in sensors via an abstraction (see architectural design).

More detail on the difference between “modern” and “classic” sensors is to be found in part 3 (field-study on sensors).

2.4.2 Integration

Another focus is integration and pluggability.

The thesis isn't about network-technology like e.g. meshing-networks, Zigbee or other “Internet Of Things”-technologies.

More important it that the system should be **capable** to integrate with different types of network-technologies when necessary.

3 Structure

3.1 Part 1 : Introduction

The chapters in this part expose a 10.000 feet-view on the project, and should respond on questions like e.g.

- Who are the stakeholders and there expectancies?
- What is the scope of the thesis/project?
- In which context is this thesis constructed?
- What was the approach of this project?
- How is this thesis structured?

3.2 Part 2: Field study

The field-study serves as glossary for the sensor-domain and summarizes the basic knowledge on sensors needed by actors using the system described in this thesis (like e.g. the characteristics sensors).

This part serves as a preparation for the remaining parts, it summarizes and expleains the most important topics. For details however it will refer to other resources (books, journals, specifications, ...).

3.3 Part 3: Requirements

Elaborates the requirements (functional and non-functional) related to the stakeholders and the actors using the solutions (and systems) described in this thesis.

3.4 Part 4: Building-blocks and architecture

Identifies and describes the building-blocks used by this thesis and synthesises/links these building-blocks into a cohesive system-architecture.

3.5 Part 5: Roll out

This parts explains how this architecture and building-blocks can be rolled to the different scenario's described in the requirements.

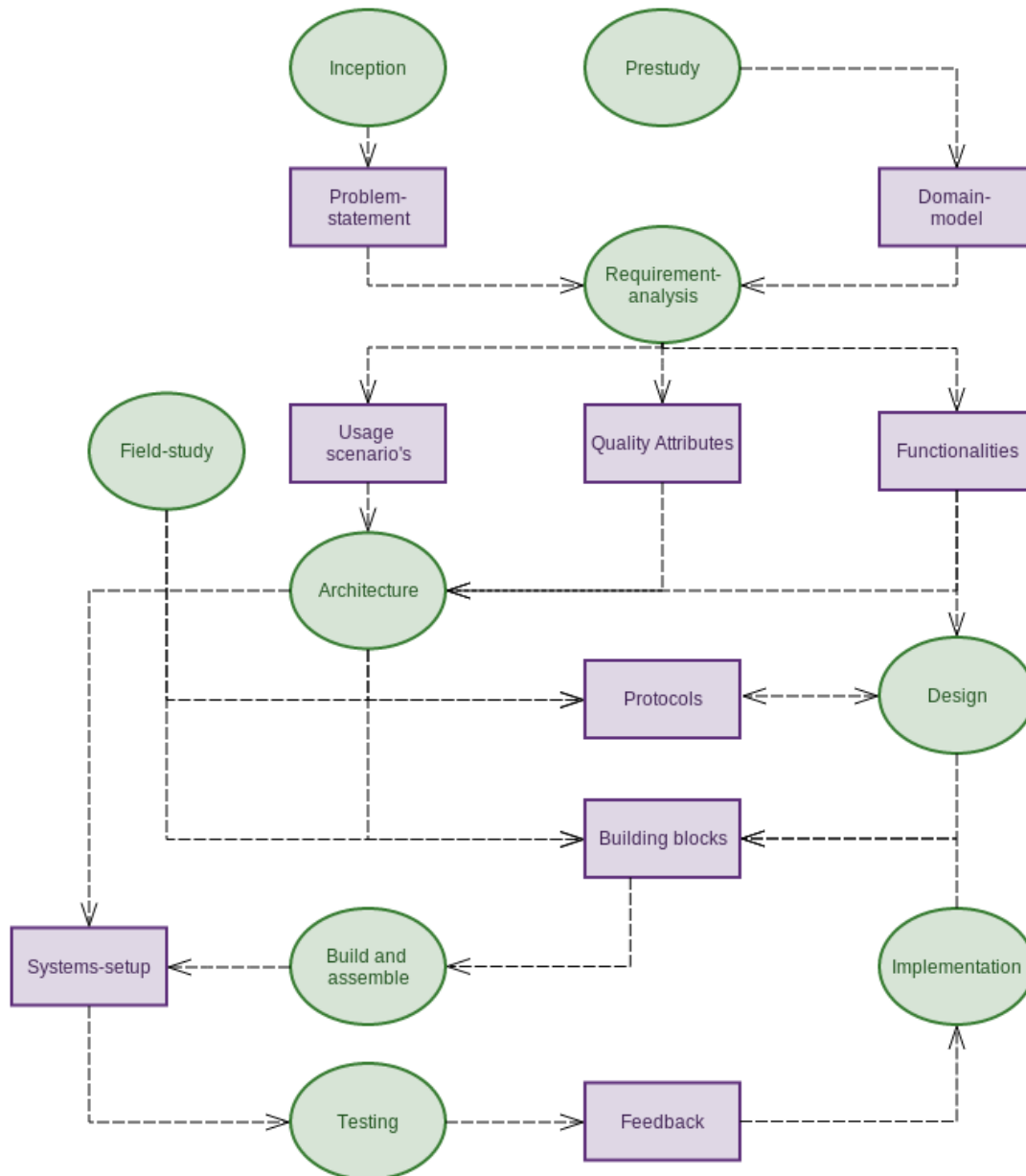
A roadmap is added to describe to what's currently have been done, what's under construction and potential directions for this solution.

3.6 Part 5: Conclusions

This part reviews the scope

4 Approach

This chapter introduces - before going into the actual content of the thesis - the **approach** (or methodology) being used for developing this thesis, its main **artefacts** and the **correlation** to the structure of this thesis.



Drawing 1: Workflow and artefacts of this thesis

4.1 Inception (part 1)

This phase defines the elements and boundaries of the project underpinning this thesis.

Stakeholders are identified, a clear **problem-statement** is structured and the boundaries are marked in a **scope-definition** based on a clear **context-description**.

These elements are incorporated in the introduction of this thesis.

4.2 Pre-study (part 2)

This phase sets the stage for further research, it contains the elements necessary to start with the design and analysis.

A **domain-model** describes the processes, core-principles and most important elements subject to this thesis.

This model should provide a **common vocabulary** (aka ubiquitous language) and reference model to be used in our analysis, design and architecture.

4.3 Requirement analysis (part 3)

In this phase we elaborate on the problem-statement to obtain a clear vision on the need and features of the stakeholders.

These requirements are expressed in the following abstractions:

- **Functionalities** and capabilities of system we want to build.
- **Usage scenario's:**
Different situation where we want to use the system
- **Quality Attributes** and non-functional requirements:
Requirements, constraints expressing criteria on how the system should operate (not what it should do). Typical examples are performance, reliability, availability, ...)

4.4 Research on what we can reuse (part 3-4)

Before starting with the design-activity a research is to be started on what techniques and components and protocols are already available for reuse.

4.5 Architectural design (part 4)

In the architectural process end-2-end **system-set-ups** are **designed** (taking into account the different type of requirements) .

Different **protocols and building-blocks** required for this are identified on various levels (hardware, electronics, software, networking, sensors, storage, ...) and matched towards various quality attributes and cross-cutting concerns (security, monitoring, ...). Typically these building blocks are visualised in layered views and block-diagrams.

This architectural analysis is also responsible to provide **traceability** towards the **requirements** (functional and non-functional) and translate them to clear constraints to be taken in consideration during design.

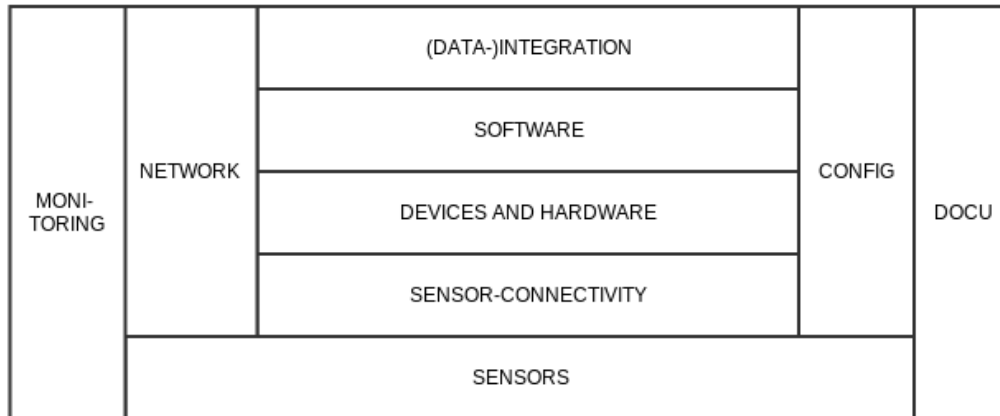
4.6 Building-blocks (part 4)

Building-blocks (or components) and **protocols** are elaborated during the design-activity. Depending on the building-block-category and specific context within this thesis these are described as:

- Proof-of-concept and other experiments used to confirm assumptions
- Typical software-design-artifacts like e.g. Component-diagrams, UML-diagrams, API-design, data-models ...
- Platform-documentation and driver
- Code
- CAD-diagrams and other electrical designs
- Statemachines
- Device-selections, comparisons and necessary argumentation
- Networks and integration-technologies
- ...

Important to state is the choice for **iterative** development, feedback and communication are bidirectional between design and implementation, a continuous loop-cycles of architecture, design, coding, testing ...

These building-blocks and implementations fall in one of the following categories or layers indicated in the diagram below:



4.6.1 Sensors

The basis project is the selection, usage and configuration of **sensors**, sensors can be compared based on specific functionalities and characteristics (described later in this text).

4.6.2 Sensor-connectivity

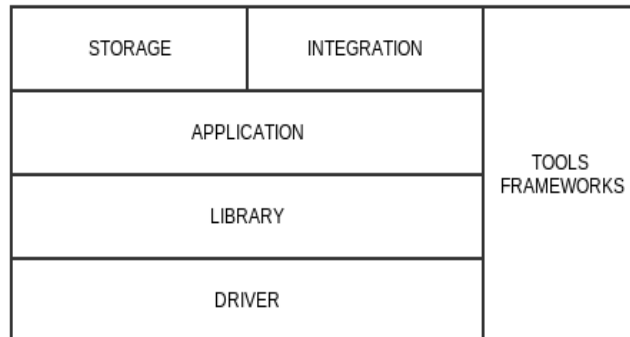
Sensors need to be **connected** via specific protocols (in most cases these are industry-standard protocols like e.g. i2c, spi, usb, ...) to a processing device.

4.6.3 Devices and hardware

These processing-**device** might be a micro-controller (e.g. AVR, PIC, ...) or a low level embedded device (e.g. BeagleBoard, Raspberry Pi, ...) and it might require some extra peripheral **hardware** to provide power, configuration, connectivity...

To activate and control these devices and their peripherals **software** is be deployed, this software will interact with the hardware, acquire and process data and provide further **(data-)integration** towards the end-user.

4.6.4 Software



4.6.4.1 Driver

The lowest level is **driver**-code (or binaries), allowing the libraries to connect to the sensors and/or other devices.

4.6.4.2 Libraries

Libraries are encapsulating functionalities like sensor-connectivity and storage behind a clear api and allow to code and deploy without know the exact detail of implementation (RTL-logic, communication-protocols).

These libraries still require the end-user to know how to code and compile on the specific platform (MCU or computer).

4.6.4.3 Application

An **application** is a software deployed on a specific device, operating system or application server to perform a specific task.

This can be an small c-program running on a micro-controller looping and capturing data from a sensor, a distributed system of components working together (and not necessarily running on the same hardware) or a html-client-application running on the user's computer.

4.6.4.4 Tools and frameworks

Naturally we try to build further on existing (open) **tools** to perform a variety of tasks (build, monitoring, ...) or reuse existing **libraries** and **frameworks**.

4.6.4.5 Storage

The applications need in general a way to **store** data on the system like e.g.

- Configuration of sensor or application
- Result of measurements
- ...

The way how we store data depends strongly on the platform, operating system (or lack of), requirements and are thus considered separated building blocks.

4.6.4.6 Integration

Software (and hardware) is to be **integrated** on a platform, components are to linked through protocols or network-technologies...

Technology or approaches to be integrated are documented as building-blocks.

4.6.5 Configuration and documentation

Configuration can be performed at all levels:

- Hardware (jumpers, switches, ...)
- Operating system (drivers, initialization-files, ...)
- Build-parameters (e.g. instructions for a build-server, ...)
- Software (setup-parameters in a file or database, connection-settings)
- ...

This configuration and the different possibilities are to be **documented** in a structured way for the end-users.

4.7 Build and assemble (part 5)

Build and assemble” ensures the delivery of the solution and products, it concerns activities like e.g.:

- Preparation of the sandbox-environment
- Documentation-structure for the different technologies
- Build-automation
- Deployment and rollout-processes
- Configuration
- User- and development guides
- Development-tools
- Operational setup (e.g. health-monitoring, operation-instructions, ...)
- ...

The thesis will provide a high-level overview of these techniques and rationale why to use specific techniques.

However most of the result and documentation will be delivered as appendixes referenced from this thesis.

PART 2 : Field-study

5 Sensors (field-study)

5.1 Introduction and structure

The goals of this chapter are to

- understand high-level the process of sensing and its the typical components
- overview of sensor-characteristics
- provide definitions and builds a common vocabulary

applied to the world of sensors and in further extension Sensor networks.

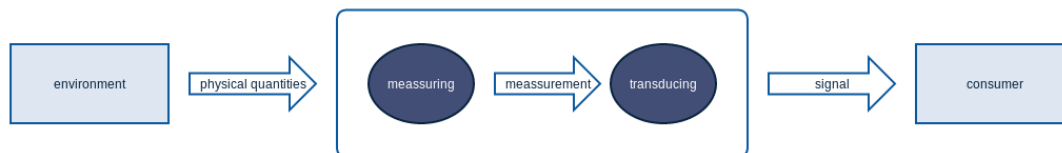
5.2 What is a sensor?

A device that receives a stimulus and responds with an electrical signal (Fraden 2010), it measures phenomena of the physical world.

These phenomena can be things you see, like light, gases, water vapor, and so on, they can also be things you feel, like temperature, electricity, water, wind, and so on. (Bell 2013)

In other words, sensors sample these physical variables and turn them into a proportional electric signal (voltage, current, digital, and so on).

Sensors can be used to measure or detect a vast variety of physical, chemical, and biological quantities, including air pressure, temperature ,chemicals, gases, light intensity, motion, position, sound and many others, as shown below.



These measurements are converted by a transducer into a signal that represents the quantity of interest to an observer or to the external world. (McGrath & Scanail 2014)

5.3 Sensor-characteristics

To work with sensors (whether it's in industry, hobby or academics) we need to understand sensors (and how sensors that measure the same measurement can differ).

In practice this is done by reading and interpreting datasheets (or specifications or just specs) delivered by the producer of these sensors (in much the same way as electronic devices).

However there is a lack of standard definition for many of these characteristics.

On top of that different sensor-types (in the same of a different domain) have different names for these characteristics.

Goal of this chapter is to give some guidance and structure around these characteristics and to develop a basic understanding of the most common features.

These characteristics should also be considered a vocabulary referenced from in the remaining chapters of this document.

Sensor characteristics can be categorized/classified as systematic, statistical, or dynamic. (Bently 1995) defines

- **systematic** characteristics as “those which can be exactly quantified by mathematical or graphical means;”
- **statistical** characteristics as “those which cannot be exactly quantified;”
- **dynamic** characteristics as “the ways in which an element responds to sudden input changes”

The following section will describe these characteristics, using their most common names, and will reference alternative names where relevant.

In most cases this inspired on the book (McGrath & Scanail 2014) and (Bently 1995)

5.3.1 Range(s)

Range(s) are static characteristic they describe both the lower and upper limits of the input or output of a sensor.

This term is commonly used in the following ways inside datasheets:

Full-scale range describes the maximum and minimum values of a measured property.

Full-scale output (FSO) is the algebraic difference between the output signals measured at maximum input stimulus and the minimum input stimulus.

Full-scale input or more commonly called **span** (or dynamic range) describes the maximum and minimum input values that can be applied to a sensor without causing an unacceptable level of inaccuracy.

e.g. IR Range sensor measures distance between 10 and 80 cm

5.3.2 Linearity

Non-linearity, which is often called linearity in datasheets, is the difference between the actual line and ideal straight line. As non-linearity may vary along the input-output plot, a single value, called maximum non-linearity, is used to describe this characteristic in datasheets.

Maximum non-linearity is typically expressed as a percentage of span. Non-linearity can often be affected by environmental changes, such as temperature, vibration, acoustic noise level, and humidity. It is important to be aware of the environmental conditions under which non-linearity is defined in the datasheet, particularly if they differ from the application operating environment.

5.3.3 Electrical specifications

Operating voltage is a range describing the minimum and maximum input voltages that can be used to operate a sensor.

Applying an input voltage outside of this range may permanently damage the sensor.

Almost all integrated circuits (ICs) have at least two pins that connect to the power rails of the circuit in which they are installed. These are known as the power-supply pins. However, the labelling of the pins varies by IC family and manufacturer.

Supply voltage - commonly abbreviated as VDD (mos-technology) based or VCC (BJT-technology) - being the voltage to be obtained from a power source for operation of a sensor (or other device).

Input Current – indication of minimum and maximum current

Power consumption – Electric power, like mechanical power, is the rate of doing work, measured in watts.

5.3.4 Speed and performance

The time taken by a sensor to arrive at a stable value is the **response time**.

It is generally expressed as the time at which the output reaches a certain percentage (for instance 95%) of its final value, in response to a stepped change of the input.

The **recovery time** is defined in a similar way but conversely.

In many cases the performance depends on the required quality of the measurements accuracy and precision for the measurement.

5.3.5 Accuracy



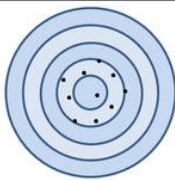
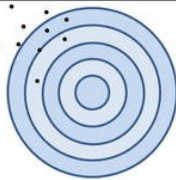
Accuracy refers to a sensor's ability to provide an output close to the true value of the measurements.

Specifically, it describes the maximum expected error between the actual and ideal output signals. Accuracy is often described relative to the sensor span. For example, a thermometer might be guaranteed to be accurate to within five percent of the span. As accuracy is relative to the true value of the measurements, it can be quantified as a percentage relative error using the following equation:

$$\text{Percentage Relative Error} = ((\text{Measured Value} - \text{True Value}) / \text{True Value}) * 100$$

5.3.6 Precision

Precision is sometimes confused with accuracy.

		Accuracy	
		Accurate	Not Accurate
Precision	Precise		
	Not Precise		

Precision describes the ability of an output to be constantly reproduced.

It is therefore possible to have a very accurate sensor that is imprecise (a thermometer that reports temperatures between 62–64° F for an input of 63° F), or a very precise sensor that is inaccurate (a thermometer that always reports a temperature of 70° F for an input of 63° F). As precision relates to the reproducibility of a measure, it can be quantified as percentage standard deviation using the following equation:

$$\text{Percentage Standard Deviation} = (\text{Standard Deviation}/\text{mean}) * 100$$

5.3.7 Resolution

Resolution, also called discrimination, is the smallest increment of the measurements that causes a detectable change in output. The resolution of modern sensors varies

considerably, so is important to understand the resolution required for an application before selecting a sensor. If the sensor resolution is too low for the application, subtle changes in the measurements may not be detected. However, a sensor whose resolution is too high for the application is needlessly expensive. Threshold is the name used to describe resolution if the increment is measured from zero, although it is more commonly described as the minimum measurement required to trigger a measurable change in output from zero.

5.3.8 Drift

If the output signal slowly changes independent of the measured property.

It is often associated with electronic age of components or reference standards in the sensor.

5.3.9 Long term drift

A slow degradation of sensor properties over a long period of time.

5.3.10 Environmental conditions

Sensor outputs can be affected by external environmental inputs as well as the sensor and itself. These inputs can change the behaviour of the sensor, thus affecting the range, sensitivity, resolution, and offset of the sensor. Datasheets specify these characteristics under controlled conditions (such as fixed temperature and humidity, fixed input voltage, and so on). If a sensor is to be operated outside these conditions, it is highly recommended to recalibrate the sensor under the conditions in which it will be used.

5.3.11 Transfer functions

Sensor characteristics describe the relationship between the measurand and the electrical output signal.

This relationship can be represented as a table of values, a graph, or a mathematical formula.

Expensive sensors that are individually calibrated may even provide a certified calibration curve. If this relationship is time-invariant, it is called the sensor transfer function. A formula for this transfer function is typically expressed as “ $S = F(x)$ ” where x is the measurement and S is the signal output produced by the sensor.

It is however rare to find a transfer function that can be completely described by a single formula, so functional approximations of the actual transfer function are used.

5.3.12 Heat and power-dissipation

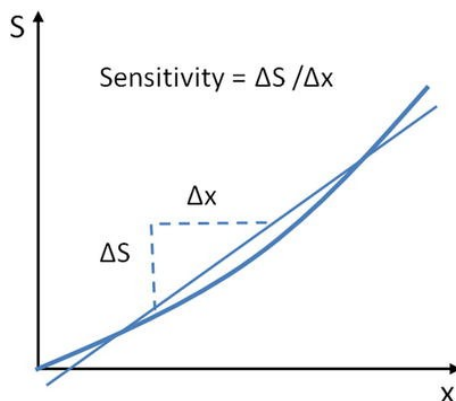
The heat that a sensor can dissipate depending on the packaging, performance and environment is an important parameter to take into account.

5.3.13 Sensitivity

Sensitivity is the change in input required to generate a unit change in output.

If the sensor response is linear, sensitivity will be constant over the range of the sensor and is equal to the slope of the straight-line plot (as shown in Figure TODO).

An ideal sensor will have significant and constant sensitivity.



If the sensor response is non-linear, sensitivity will vary over the sensor range and can be found by calculating the derivative of S with respect to x (dS/Dx).

5.3.14 Compatibility

The (digital) interface for sensors coming the same supplier can be compliant with each other.

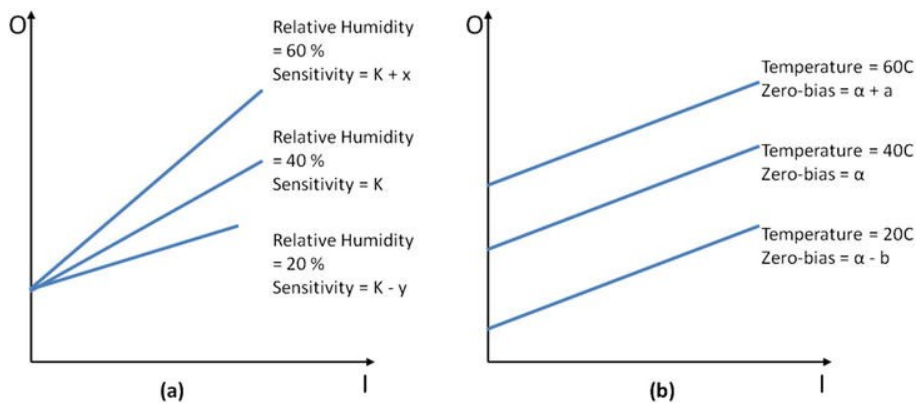
5.3.15 Interfering Inputs vs modifying inputs

Modifying inputs changes the linear sensitivity of a sensor.

The voltage supplied to the sensor, V_s , is a common example of a modifying input as it can modify the output range of the sensor, which in turn modifies the resolution and sensitivity.

Interfering inputs change the straight-line intercept of a sensor.

An example of a temperature effect on sensor output is shown below:



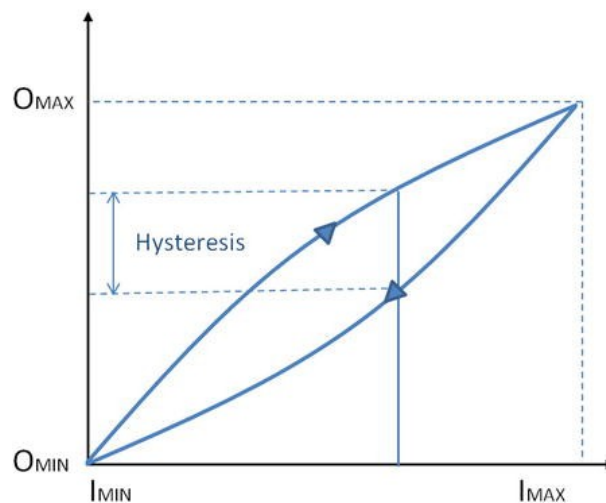
(a) Effect of a modifying input (relative humidity) on the sensor's sensitivity (K)

(b) Effect of an interfering input (temperature) on the sensor

5.3.16 Hysteresis

The output of a sensor may be different for a given input, depending on whether the input is increasing or decreasing.

This phenomenon is known as hysteresis and can be described as the difference in output between the rising and falling output values for a given input as illustrated in the figure below copied from the book (McGrath & Scanail 2014) .



Like non-linearity, hysteresis varies along the input-output plot; thus maximum hysteresis is used to describe the characteristic. This value is usually expressed as a percentage of the sensor span. Hysteresis commonly occurs when a sensing technique relies on the stressing of a particular material (as with strain gauges). Elastic and magnetic circuits may never return to their original start position after repeated use. This can lead to an unknown offset over time and can therefore affect the transfer function for that device.

5.3.17 Error

Systematic errors are reproducible inaccuracies that can be corrected with compensation methods, such as feedback, filtering, and calibration (Wilson 2004)

These errors result from a variety of factors including interfering inputs, modifying inputs, signal attenuation, signal loss, ...

Random error (also called noise) is a signal component that carries no information. The quality of a signal is expressed quantitatively as the signal-to-noise ratio (SNR), which is the ratio of the true signal amplitude to the standard deviation of the noise. A high SNR represents high signal quality.

5.4 Evolution of sensors

In the previous chapter the most common characteristics used in the sensor-domain are summarized to give the reader the ability to interpret and understand most of the data-sheets used for sensors.

This thesis however (due to the context of the research it supports) focusses on the interaction with modern (smart) sensors (and in most cases also digital).

5.4.1 Smart sensors

When you work with **classic sensors** engineers linearise non-linear sensor output using hardware or software to leverage the advantages of linear sensors.

The traditional hardware-based linearisation method often requires manual calibration and precision resistors to achieve the desired accuracy.

In many cases this requires deep understanding of the characteristics and electric properties of these devices.

Modern smart sensors on the other hand employ less complex and less costly digital techniques to create a linear output.

These techniques perform digital linearisation and calibration by leveraging the smart sensor's integrated micro-controller and memory to store the factory calibration results for each individual sensor.

The micro-controller can correct the sensor output by searching for the compensation value or the actual linearized output in a look-up table.

If memory is limited, calibration coefficients, rather than a full-look up table, are used to construct a linearized output.

5.4.2 Connection-protocols (for smart sensors)

A connection-protocol defines how a sensor talks to the processing unit (like an MCU or other device supporting low level connectivity).

The protocol defines how the wires should be connected and how your code should ask for measurements. (Inspired on the book (Tero Karvinen & Valtokari 2014))

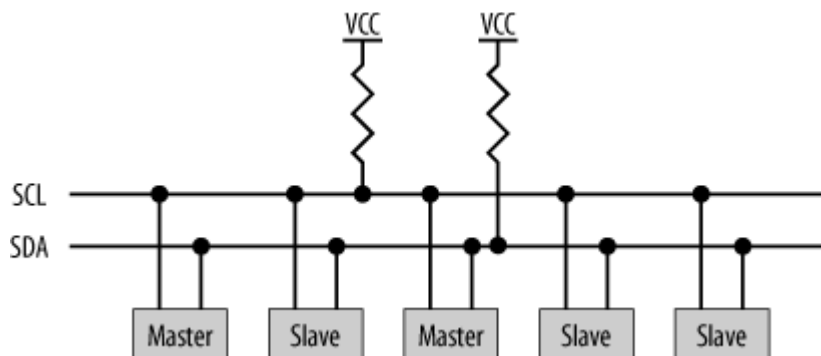
There is a huge variety of different types of sensors, luckily enough the number of popular protocols is more limited.

5.4.2.1 I2C

I2C (Inter-Integrated Circuit) bus is a very cheap yet effective network used to inter-connect peripheral devices within small-scale embedded systems.

Several manufacturers, such as Microchip, Philips, Intel, and others produce small micro-controllers with I2C built in.

It uses two wires to connect multiple devices in a multi-drop bus. I2C allows 128 devices to be connected to the same wires.



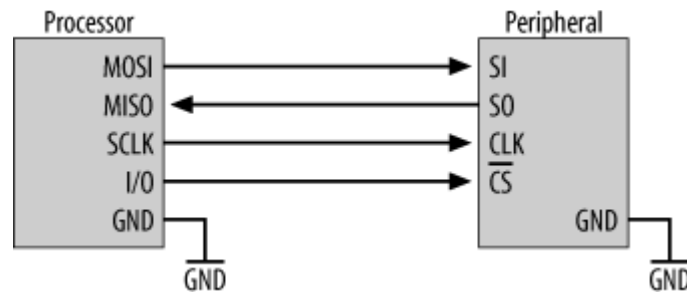
The two wires used to interconnect with I2C are **SDA** (serial data) and **SCL** (serial clock). Both lines are **open-drain**. They are connected to a positive supply via a **pull-up resistor** and therefore remain high when not in use.

Devices may be attached or detached from the I2C bus without affecting other devices.

5.4.2.2 SPI

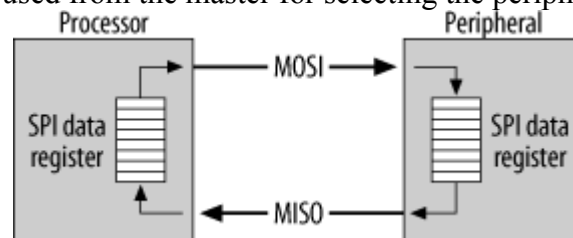
SPI is another industry standard protocol.

The Serial Peripheral Interface (known as SPI) was developed by Motorola to provide a low-cost and simple interface between microcontrollers and peripheral chips. (SPI is sometimes also known as a four-wire interface)



SPI uses four main signals:

- Master Out Slave In (MOSI),
- Master In Slave Out (MISO)
- Serial CLoCK (SCLK or SCK)
- Chip Select used from the master for selecting the peripheral.



Both masters and slaves contain a serial shift register. The master starts a transfer of a byte by writing it to its SPI shift register.

As the register transmits the byte to the slave on the MOSI signal line, the slave transfers the contents of its shift register back to the master on the MISO signal line. In this way, the contents of the two shift registers are exchanged. Both a write and a read operation are performed with the slave simultaneously.

SPI can therefore be a very efficient protocol.

5.4.2.3 Digital Resistance (ON/OFF)

Some sensors work like a button and have two states, on or off. These sensors are easy to read. The on state is represented when a voltage referred to as HIGH is applied to the MCU input pin.

This is usually either 3.3 volts or 5 volts depending on the set-up you're using.

5.4.2.4 Serial Port (UART)

A serial port sends text characters between two devices.

It's the same technique your computer uses when talking to a micro-controller over an ftdi-breakout.

5.4.2.5 Bit-Banging

Sometimes, a sensor is unusual enough that a standard protocol won't work with it. In those cases, you need to craft up your own code to talk to that sensor.

This is often called bit-banging, because you're manipulating the signal from the sensor often at the bit level.

More information and detail on these techniques can be found inside books like (Catsoulis 2009) and (Williams 2014a).

5.4.3 Sensor-networks

A new evolution that we see emerging is the usage of sensor-networks.

A sensor network consists of a **group** of **smart sensors** that are wired or wireless connected to another node or to a common **aggregator**.

In networking terminology, each component in the network that has a communications module is called a node.

5.4.3.1 Sensor-node

At the lowest (or *leaf*) level of the sensor network is a basic sensor node. This is the type of node described thus far—it has a single sensor and a communication mechanism. These nodes don't store or manipulate the captured data in any way—they simply pass the data to another node in the network.

5.4.3.2 Data node

The next type of node is a data node. Data nodes are sensor nodes that store data. These nodes may send the data to another node, but might also work in an isolated mode where they store data locally to a storage mechanism like a data-card, a database or stream their output directly to a visual output like an LCD-screen, led-indicators, ... Data nodes can be used to form autonomous or unattended sensor networks that record data for later archiving.

In part 4 these kind of nodes will be referred upon as **sensor-agents**

5.4.3.3 Aggregation-node

Another type of node is an aggregate node. These nodes typically employ a communication device and a recording device (or gateway) and no sensors. They're used to collect data from one or more data or sensor nodes. In the examples discussed thus far, the monitoring system would have one or more aggregator nodes to read the data from the sensors.

In part 4 these kind of nodes will be referred upon as **sensor-hub**

5.4.4 *Sensor-networks in the context of this project*

Integrating sensors makes sense in the context of this project.

The primary scenario covered in this thesis requires data to be gathered coming from different sensors on different locations inside a building

This data is then to be collected, aggregated and stored in one processing device.

The integration-protocols used (today) for digital (or smart) sensors – in most cases i2c or spi - are not very well suited to connect (in reliable way) sensors over a long distance. In essence these protocols are created for on-chip- or on-board-communication.

For i2c e.g. you might solve the problem by

- using i2c-extendors
- lowering the clock-rate of i2c

But on the other hand the availability, cost and efficiency of current network-technologies outweighs these efforts.

5.5 Sensors in the domain of Building Physics

5.5.1 *Relative humidity*

Humidity describes the quantity of water vapor in a gas like air.

There are many different ways to express humidity, e.g. relative humidity, absolute humidity, dew point temperature or mixing ratio.

Absolute humidity is the mass of water vapor divided by the mass of dry air in a volume of air at a given temperature. The hotter the air is, the more water it can contain.

Relative humidity is the ratio of the current absolute humidity to the highest possible absolute humidity (which depends on the current air temperature).

A reading of 100 percent relative humidity means that the air is totally saturated with water vapor and cannot hold any more causing condense or rain.

5.5.2 *Temperature*

Sensors (like the SHT21 from Sensirion) - that are used in the context of the research - measure both temperature and relative humidity.

This makes sense because relative humidity is analysed in the context of environmental parameters like temperature and differential pressure.

5.5.3 *Differential pressure*

Differential pressure is the difference in pressure between two separate points.

Differential pressure can be measured between two points on independent systems or between two different points on the same system.

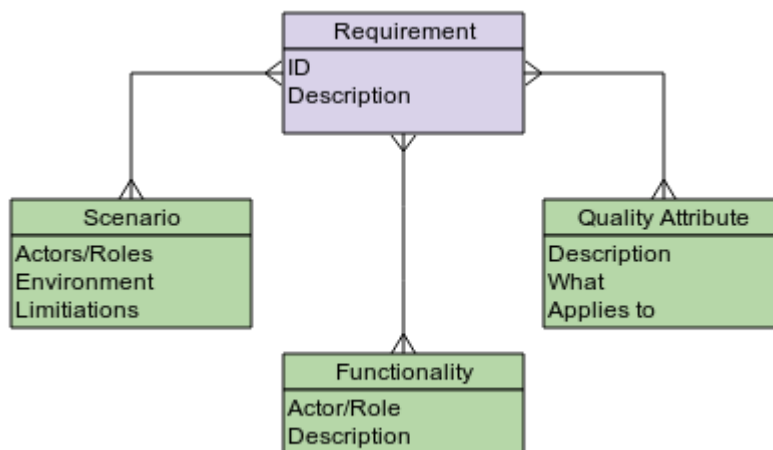
The main use for a differential pressure sensor is to measure the difference in fluid or gas pressure across a restriction in a pipe.

PART 3 : Requirements

6 Requirements and concerns of the system

6.1 Introduction and structure

This chapter aims at describing what the in detail what is expected from the point-of-view of the stakeholders **what** the system should do.



There are 3 **types** of requirements described in this chapter:

- **Scenario's** describing for one (or more) **actors** (and/or **roles**) how the system should be used.
These scenario's exists in a specific **environment** (e.g. public building, environment, ...) and can bring along specific **limitations** (e.g. no network-power, low or high/temperatures,) ...
- **Functionalities** describing what specific **actors** (and/or **roles**) need to execute their specific activities
- **Quality attributes** (and or concerns) can be considered non-functional requirement. For each quality attribute the text explains the (**what-part**) and described how this is **applied** to the sensor-domain and -research.

6.2 System usage (scenario's and actors)

6.2.1 Actors/roles

In the context of this thesis research towards energy-efficiency in buildings) there are 2 primary actors (or roles) using this system:

- Students in the field of Building Physics
- Researchers

Outside the context of the thesis we can also consider the other types of actors interacting with the system:

- Hobbyists and DIY-community
- Developers (soft- and hardware) interested in open-source development

In some or many cases these roles overlap, it's important however to make the distinction in the usage-patterns implying other functional and non-functional requirements.

6.2.2 Education and classrooms

Context and roles

In the context of their studies (building physics) the students need to execute test and lab-exercises.

They need an easy way to measure, monitor and record the result of their experiments with a variety of sensors.

On the other hand they might also participate (now or in the future) in academic research and need to be acquainted with the characteristics and limitations of these sensors in a larger scale-environment.

Environment

In most cases this is a classical environment (lab) but also in a study environment or at home.

Limitations and expectancies

The users are assumed to have a **basic** knowledge about software and electronics:

- Invoke Basic command-line instructions (e.g. start scripts and curl-commands)
- Invoke a intuitive and well-documented library in high-level language like Java
- Assemble/construct very simple circuits on a breadboard (in the worst case) based on a schema

They are however not expected to be capable to:

- write applications connecting to sensors from the bottom up and know the details of low-level communication-protocols like i2c, spi, ...
- experts in electronics

6.2.3 *Specialized building accommodation for academic research*

Context and roles

Academic research inside the infrastructure of the KUL as described in the introduction of this thesis.

Researchers have at their disposal a specific HVAC-infrastructure where they can trigger specific sequences environmental manipulations (e.g. manipulate heating, humidity, ..),

Environment

The specific case referred to is a building in Gent with specific HVAC-infrastructure.

A electric and network-infrastructure is foreseen, it possible to provide Ethernet and DC-power.

Limitations and expectancies

The same assumptions as the “education”-scenario's are applicable here

Different here is that the researchers are expected to have a good understanding of the sensor-characteristics as they are documented in the field-study-part.

They need to be able to adjust and adapt their observations to limitations of the sensors (e.g. reaction-time, maximum ranges, ...)

6.2.4 *Other academic research on buildings*

Context and roles

Same roles but in this case the research is done in buildings or constructions not customized or specialized towards the research.

Environment

All kind of buildings, in many cases large public buildings like churches, train-stations, ...

Limitations and expectancies

Research can be done on large scale, up to 1000 points to be monitored. We **cannot assume** that (wired) network-connectivity is foreseen everywhere. Also basic electric infrastructure might miss so the devices are supposed to be powered by battery.

6.3 Functional requirements

6.3.1 Controlling sensors at runtime

The actors need to be able to control sensors:

- Trigger **measurements** a periodic intervals and change the duration of this intervals
- **Configure** the **attributes** of the **sensors**, typical example is that you might want to choose the measurement-resolution for a “relative humidity”-sensor.

It should be possible to trigger these actions be done at **runtime**, no extra programming is to be performed to achieve this.

This “runtime-configuration” should be

- **accessible** via an **open** protocol like a command-line or webservices (e.g. REST-protocol)
- the user can **query** this system for **attributes** that can be configured on the available sensors (and possible minimum-maximum values)

6.3.2 Collect data from sensors

The measurements are stored and can be collected via the same runtime-interface as mentioned in the previous requirements.

The user can define in this “**query**” filter arguments like

- a period in time (from/to)
- all measurements since a specific data/time

This data is returned in a parsable text-format (e.g. csv, json, ...)

6.3.3 Correlation of measurements to configuration

The data polled from the runtime should also indicate which configuration had been used at the moment of measurement.

6.4 Concerns and quality Attributes of the system

Where the previous (functional) requirements described what the system should do, this part will describe level of quality that is expected from the system.

In software-architecture-literature the level of quality is often referred to as quality attributes, concerns or system-elements

You can look up on them as the major technical challenges of a system (software, hardware, network, ...).

Such a concern, problem, or system element is architecturally significant if it has a wide impact on the structure of the system or on its important quality properties such as performance, scalability, security, reliability, or evolvability...

6.4.1 Operability, monitoring

What:

Operability is about how the system will be controlled, managed, and monitored.

The aim of the operational viewpoint is to identify a system-wide strategy for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

Monitorability deals with the ability of the users (or depending on the context an operations staff) to monitor the system while it is executing.

Items such as queue lengths, average transaction processing time, and the health of various components should be visible to the operations staff so that they can take corrective action in case of potential problems.

Scenarios will deal with a potential problem and its visibility to the operator, and potential corrective action.

Applies to:

Errors detected in activation or measurement sensors

The system should keep track of the state of the sensors e.g.

- Last time of connection and measurement
- Errors reported during connection, measurement or configuration

The system should expose a view on the current errors occurred and their status via an interface that you can integrate in typical monitoring-tools like e.g. Nagios (example command line or web service).

Logging and error handling

The software interacting directly with the sensors should not absorb error-information.

Each error that is known and documented in the datasheet or other official documentation is to be handled appropriate and documented in the software-documentation

All error (expected or not) are to be logged and centrally managed (see also scenario's) pushed to a central location.

6.4.2 Maintainability and modifiability

What

Modifiability is about change, with the focus on the cost and risk of making changes. (Bass et al. 2013)

Typically the following questions are raised:

- What can change?
- What is the probability of a change?
- What is the cost of changes?

Applies to:

Replacing components:

The software should be able to be deployed on a variety of devices by isolating the system-behaviour in the software-libraries and -components.

Adding new types sensors:

For new experiments or measurement-results we might require new types of sensor to be integrated.

Adding support for new types sensors of sensors should be just a matter of implementing an interface or header file.

The developer - adding support for a new sensor - should not worry about low-level system details.

Documentation on design and code quality

The design-principles are to be documented so that a user and/or maintainer-perspective can understand the core-principles of the system.

The code should also be self-documenting.

This can be achieved by

- modularity
- careful choosing naming of interfaces, methods, classes, modules...
- unit-tests on the code should be also self-documenting so that

More info on code quality can be found at (Martin 2009)

More info on test-driven-development can be found at (Matt Stephens 2010) and (Grenning 2011)

More info on modularity can be found at (Knoernschild 2012) and (Grenning 2011)

6.4.3 Configurability and flexibility

What:

The way in which a software system is set up, the level parameters that can be modified and in what phase in the software-lifecycle these can be applied (e.g. at compile-time, packaging, deploying, runtime ...)

A attribute linked to that is the ability of the system to be flexible in the face of (inevitable) change that all systems experience after deployment.

In other words the whole of concerns related to dealing with this change during the lifetime of a system and the balancing towards cost and effort.

Applies to:

Runtime parametrization of sensors

The configuration for sensors should be modifiable at runtime.

The user should have a simple non-graphical-system for querying and modifying these attributes that you can easily integrate (command line, REST-service, SOAP, ...)

Convention over configuration

Convention over configuration a concept that systems, libraries, and frameworks should assume reasonable defaults. Without requiring unnecessary configuration, systems should "just work" with a minimum of configuration.

e.g. The address used to connect via i2c is for many sensors configurable, the software should connect by default on this port but provide the ability to reconfigure this when needed.

Adding support for new sensors

The system should be flexible enough that new sensors can be added without the need for programming (or at least with some kind of declarative programming that no requires a detailed knowledge of the system).

6.4.4 Usability

What:

Usability is concerned with how easy it is for the user (interacting with the system) to accomplish a desired task and the kind of user support the system provides.

Applies to:

User-experience

Users with only a basic knowledge of electronics and software - need to be able to configure and set up sensors in order to interact with them.

This includes different levels of support:

- Software-libraries to enable easy interaction with sensors connected on different kind of devices
- Software-applications command-line or web tools
- Documentation on how to set up the hardware and deploy the software
- ...

Beside the software and the necessary drivers all necessary instructions and pre-requisites should be available.

6.4.5 Hierachy/Modularity/Regularity/Encapsulation

What:

Hierarchy involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.

Modularity states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects.

Regularity seeks **uniformity** among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

Applies to:

System-abstractions

The software needs to run in a variety of system-architectures and -configurations this means that we should (wherever possible) isolate system-capabilities like e.g.

- gpio-connectivity
- digital protocols like i2c, spi, uart
- ...

into separate interchangeable modules.

Interchangeable capabilities

Capabilities like logging, messaging, storage should be interchangeable.

Different variations for these capabilities with different qualities and costs might be chosen for different scenario's.

Sensor-encapsulation

The code handling with sensors for configuration and measurements should be isolated away in one class (or module in c).

Sensor-abstraction

All sensor-implementations should expose/implement a generic interface (header-file in c) allowing other layers (like coordination-modules, storage, logging, ...) to interact with sensors via this same (uniform) interface.

6.4.6 Reliability, availability, durability, autonomy, resilience

What:

Availability refers to a property of a system that it is there and ready to carry out its task when required (Bass et al. 2013).

Availability builds upon the concept of **reliability** by adding the notion of recovery that is, when the system breaks, it repairs itself and handles failures that could affect (Rozanski & Woods 2012).

Availability quite broad and closely related to many other quality attributes like e.g. security, autonomy, stability, performance indicating the risk for a system to become available (e.g. in the case of security a denial of service-attack can bring a system down and mak it non-available)

Autonomy can be considered independence of control, ability of a device of system.

Resilience implies that the system is able to react on sudden events impacting the systems (e.g. load, reconnecting, ...)

Applies to:

Retries on configuration

As already documented in the monitoring-quality-attribute.

The system communicating with the sensor needs to deal as much as possible with errors it knows about (and which are recoverable).

The isolated module (see also attribute on modularity) is responsible to identify these errors and to apply retries when possible.

The system should also provide a way of configuring the number of maximum retries.

Recovery

For some of the scenarios it's might be critical that systems are able to run for a long time without that manual intervention are required (e.g. to recover from system-errors).

e.g. In case of power interruption the system should be able to start again when powers comes back up (mitigations are possible on both hard- and software-level).

6.4.7 Scalability, mobility, portability

What:

Scalability in this **context** (not the scalability in context of performance) implies that your software can be used to run in different environments and circumstances (hard- and software)

Portability refers to the ease with which software that was built to run on one platform can be changed to run on a different platform.

Applies to:

Portability of software

The software should be able to run on as much devices as possible.

This can be achieved by abstracting/isolating the system (see also the attribute modularity) and by using software-stacks that can run easily on a variety of hard- and software.

Different scenario's

The software should be scalable towards different usages.

From local testing in educational scenario's up to long term research.

6.4.8 Scalability and performance

What:

Performance is about the ability to meet timing requirements.

When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time.

The **scalability** property of a system is closely related to performance, but rather than considering how quickly the system performs its current workload, scalability focuses on the predictability and adaptability of the system's performance as the workload increases.

Two kinds of scalability are horizontal scalability and vertical scalability. Horizontal scalability (scaling out) refers to adding more resources to logical units, such as adding another server to a cluster of servers. Vertical scalability (scaling up) refers to adding more resources to a physical unit, such as adding more memory to a single computer.

Applies to:

Sensor-performance

Many datasheets (certainly for digital sensors) documents performance-characteristics for sensors. The sensor-abstraction (module responsible for interacting with a specific type of sensor, see also the modularity-part) should be able to protect the user from configuring wrong values.

Support for different devices

As discussed in the modularity-part the system should be able run on different types of systems allowing scaling up. In further extent we should also be able to distribute different concepts like connecting to a sensor and processing the data to different devices. e.g. Migrating the data-processing from a Raspberry Pi to a bigger sized server can only be reached if you modularize and separate this data-processing-part.

Scaling out

The system-design should be able to of dividing the load over different modules. This can typically be reached by working with messaging systems and other non-blocking systems (see also <http://www.reactivemanifesto.org/>)

6.4.9 Identification, correlation and timing

What:

Identification (in this context) is about attributing an identity to the different components of the system (like sensors, processing devices, ...)

Correlation (in this context) is about linking different measurements based on **timing** and other attributes.

Applies to:

Attributing identity to sensors

Each sensor - integrated into the system - should be attributed with a unique identifier. This is the responsibility of the system but the user should also allow users of attributing this identifiers manually.

Different strategies should be enabled to create such identifier (e.g. combination of serial-number and type of the sensor since many smart sensors offer this number via their digital interface)

Correlation between measurements and configuration

Configuration applied on a sensor connected to the system is to be kept in storage.

When the user queries for (requests a report) on these measurements he should be able to verify which measurements were executed against which configuration.

Time-synchronisation

When a system collects over multiple (or a group of) sensors measurement-data it will store this data in storage for further analysis.,

This system might also be used in the context of sensor-networks (with ethernet, wifi, zigbee, ...) and thus on different devices.

For this reason the system should foresee (or at least delegate to another system)

- Time-synchronisation over the different devices
- Monitoring for deviations on timing between the measurement-device and centralized storage (if this capabilities run on other devices or systems)

See also monitoring-attribute for error-view on a system.

6.4.10 Interoperability

What:

Interoperability is about the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context. The definition includes not only having the ability to exchange data (syntactic interoperability) but also having the ability to correctly interpret the data being exchanged (semantic interoperability). (Bass et al. 2013)

Applies to:

Communication between different devices

As discussed previously in the modularity- and scalability-perspective the system can be distributed over different devices.

These devices should be able to communicate with each other over different channels.

As a consequence the software should modularize the integration-capability and support multiple protocols (depending on system- and environment-constraints).

6.4.11 Testability

What:

Software testability is the degree to which a software artefact (i.e. a software system, software module, requirements- or design document) supports testing in a given test context.

Applies to:

Testing in isolation

We should be able to test the components in isolation.

e.g. By providing stub-versions (fake-versions) of system-capabilities (like i2c) we can test the logic of the sensor-abstractions in isolation.

This allows us to provide testing in a very early stage of the (embedded) development.

This can be achieved through a high degree of modularity of the system (see modularity-attribute)

6.4.12 Security

What:

Security is the system's ability to protect data, information and functionality from unauthorized access (while still providing access to people and systems that are authorized).

Applies to:

Avoiding turtle shell security

The network-security will be for most cases isolated inside the integration-components or delegated to infrastructure-configuration (e.g. firewall).

This doesn't mean that security shouldn't be built in the systems, even when your network is protected against network-attacks, typical example is sql-injection.

See https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project for more information on different kinds of security-vulnerabilities

Protecting sensors

The sensor-abstractions (see also modularity) are responsible to protect the sensors for wrong configurations (see datasheets).

6.4.13 Open standards and open source

What:

Open source software is software that can be freely used, changed, and shared (in modified or unmodified form) by anyone. Open source software is made by many people, and distributed under licenses that comply with the Open Source Definition.

Applies to:

Avoiding vendor-locking

The system makes (as much as possible) use of open and accessible techniques to avoid vendor-locking and to provide scalability to different systems.

6.4.14 Accessibility

What:

Accessibility is the degree to which a product, device, service, or environment is available to as many people as possible. Accessibility can be viewed as the "ability to access" and benefit from some system or entity.

Applies to:

Actors:

As stated in the actor and scenario-chapter the assumption is made that the users of the system have a minimum level of software- and hardware-capabilities.

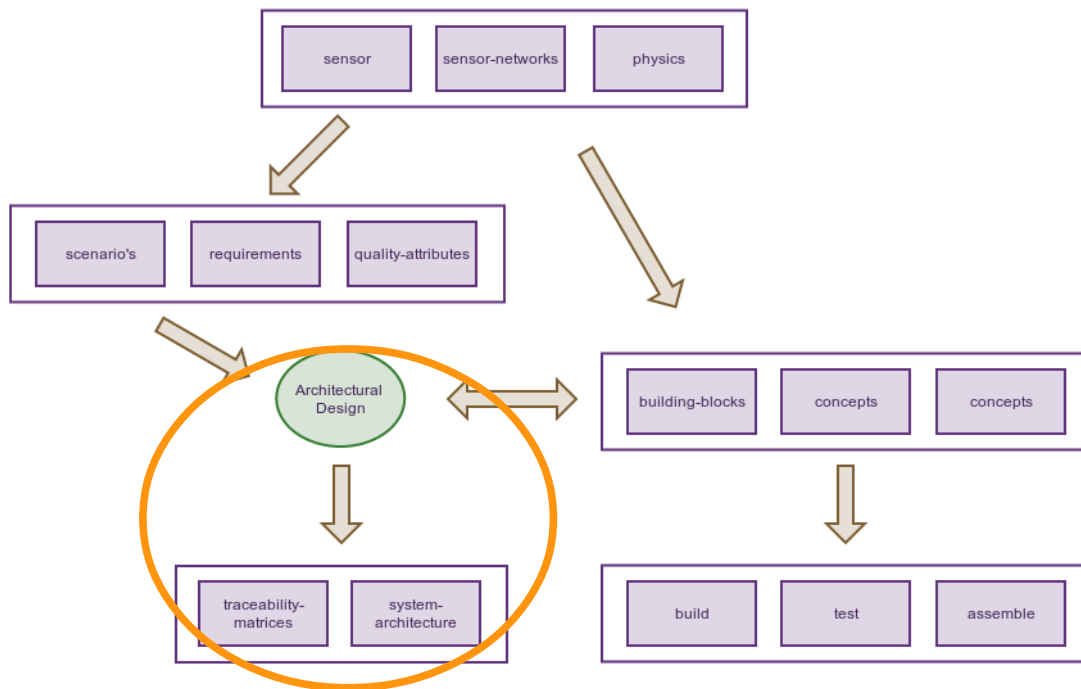
In other words they can typically work with a command-line, structured text-files (xml, json ...) and they can work with basic electronic components (resistors, capacitors, breadboards, jump-wires) when required.

The system should take this into account and should provide usability in its core design and provide detailed documentation and manuals.

PART 4: Architecture and building blocks

This part identifies initially – corresponding to the the requirements - the building-blocks, protocols and design-decisions and considerations made to satisfy the needs of the identified stakeholders.

7 Architectural Design



The chapter summarize the building blocks and layers used for this system.

Beside identifying these blocks it want to place these building blocks together in a co-herent system architecture and give enough traceability for justifying the requirements (functional and non-functional).

7.1 Summary of required system-capabilities

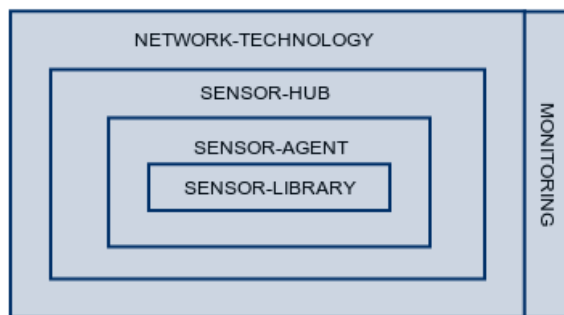
In a nutshell, the stakeholders expect from the system the following capabilities:

- Work and program with **different types** of sensors without being required to know the low-level-details of the sensors
- **Configure** the sensors with the help of this library and programming model
- **Intercepting** data from these sensors and be able to **store** it somewhere
- **Query** the measurement- and configuration-data that was intercepted via this software from a group of sensors
- **Relate** this stored **measurement**-data with the **configuration** that was used to measure from sensors.
- Check health and **monitoring** on these sensors

Beside these core system-capabilities the stakeholders require a **documentation** on

- How to work with the system and deploy the software?
- How to integrate the sensor-components, the software and different platforms with each other?

7.2 Layering of the system

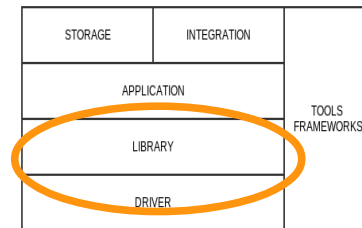


The building-blocks identified in this chapter (and elaborated in the next) are built **on top of each other**.

These building blocks (or layers) each represent distinct capabilities that are elaborated later in this part of the thesis.

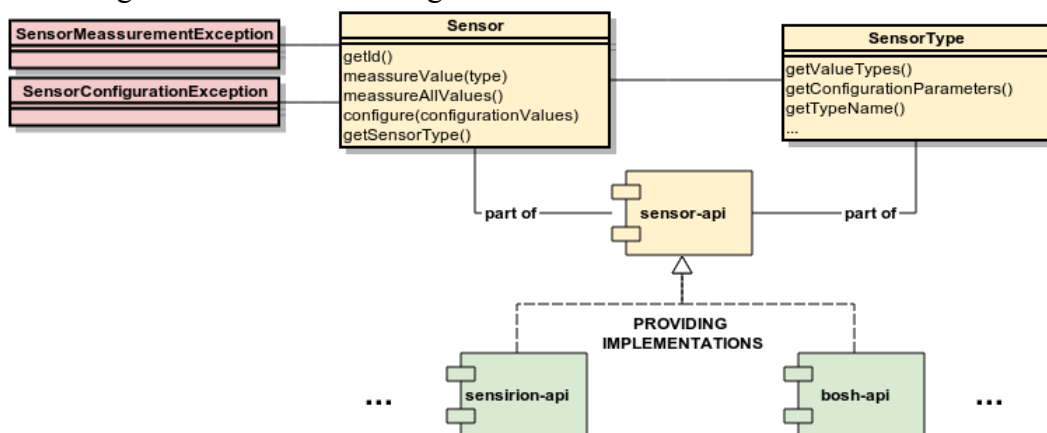
7.3 Sensor-library (and api)

The Sensor-library or api is the lowest level of software provided within the context of this thesis.



Basically it's an abstraction-layer for working with different types of sensors that exposes:

- **Sensor**-abstraction hiding away the details of interaction and communication with the Sensor
- **SensorType**-abstraction providing metadata like e.g. the type of values returned by the sensor and configuration-parameters
- **Exceptions** giving information on what went wrong in case of a problem during measurement or configuration



The 2 important advantages/capabilities of using such a model are:

- **Encapsulation**

The api hides away the connectivity and boiler-plate-code and allows programmers using this library to focus on their application.

These sensor-abstractions have also a translation-functionality, in many cases some calculations needed to be performed to obtain usable output.

- **Generic abstractions**

The interfaces (Sensor and SensorType) basically work with key-pair-values for configuring and measuring.

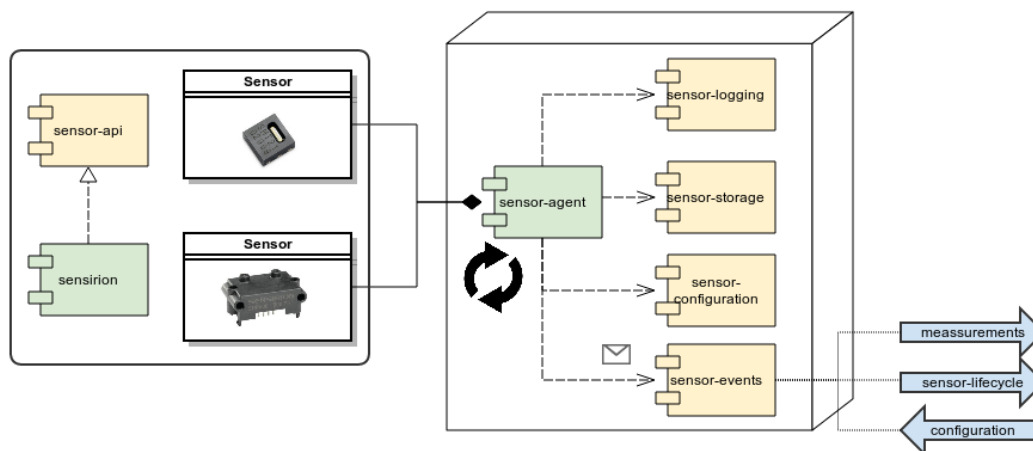
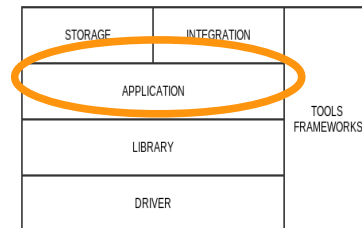
This allows higher layers (like sensor-agent and sensor-hub) to work with values without knowing the exact semantics of the sensor.

7.4 Sensor-agent

The **sensor-api** is a library, not a **ready-to-use-and-deploy** component.

If you want to use this library you still need yourself to program an application using this api, handling (and logging) the errors and integrating with some storage or output-technology.

This is where the **sensor-agent** comes in.



The sensor-agent basically it's a process that:

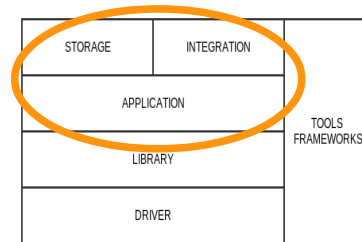
- Runs on a micro-controller or another embedded device connected to **one** or **more** sensors using the abstraction of the **sensor-api**
- Polls these sensors corresponding a specific timing for **measurements**
- Stores this output of the sensor locally via the **sensor-storage**-module
- Uses sensor-events-module to
 - Push measurement other processing devices (see sensor-hub)
 - Receive instructions to reconfigure the sensors
- Handling errors (and retrying-behaviour depending on configuration) and logging them via a logging-abstraction (**sensor-logging**)

The yellow components on the drawing - that the sensor-agent is relying onto - are **abstractions** implying that various implementations might be used depending on the context

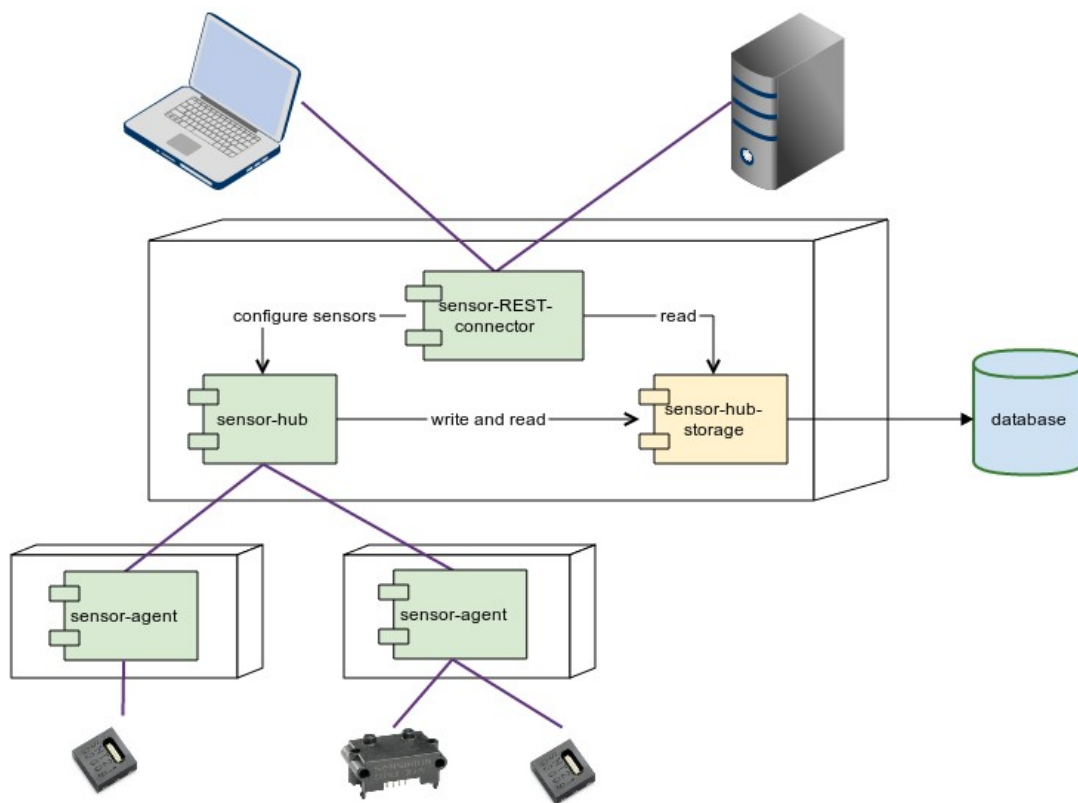
(e.g. sensor-events might be connected via uart to laptop but might also be messaging module communicating with a remote server).

7.5 Sensor-hub

The responsibility of the **sensor-agent** is limited to sensors connected to the local host-device (via protocols like i2c, spi, uart, gpio, ...)



With the **sensor-agent** we have a standardized process for polling data from sensors and to push this data to the outside world (via the sensor-events).



The **sensor-hub** is component that builds further on this capability by:

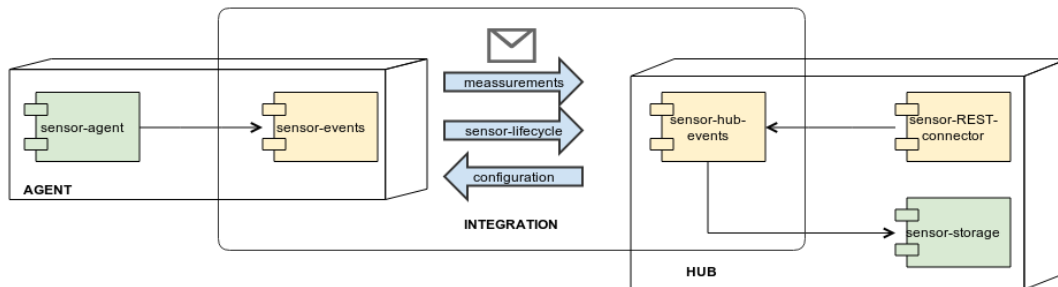
- Communicating with and listening with multiple sensor-agents
- Providing centralized storage to store data coming from the sensor-agents (measurements and configuration)
- Providing via a REST-connector (the hub runs as a webserver)
 - to read the measurements of the different sensors
 - control and configure the configurations

7.6 Integration and network

7.6.1 Messaging

The agent- and the hub-component run in most cases on different devices or different processes .

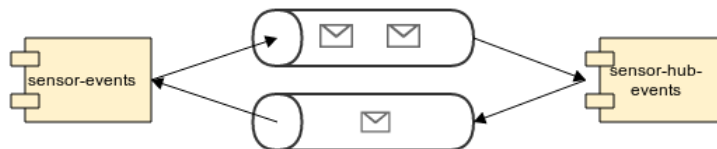
(it's possible to run the sensor-agent and sensor-hub in the same process which makes a lot of sense for unit-testing).



The modules used by these processes are the sensor-events (agent) and sensor-hub-events (hub). These modules are responsible for passing messages between 2 processes by making sure they are:

- Talking the same language (in essence same serialization-format and -model)
- Using the same communication-channel

The 2 components use a non-blocking messaging model (asynchronous communication).

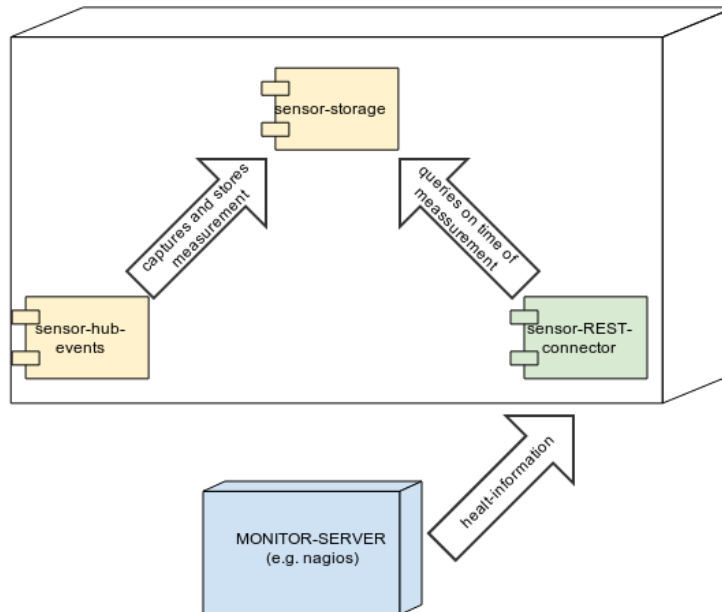


This allows both processes not to be blocked by communications and allowing the measurement- and configuration-process to proceed.

As you will see in the elaboration in the next chapter the internal flow between the agent and the hub is built around this principle.

The current implementation is based on ActiveMQ (<http://activemq.apache.org/>) but alternative implementations are under development using MQTT-SN allowing mesh-networks (Zigbee, radio frequency, ...) to be used.

7.7 Monitoring



The sensor-hub-system provides no monitoring on its own, this is left up to specialized software like Nagios and Wireshark.

It supports however support monitoring by providing (via the REST-connector) a representation of the sensor-health.

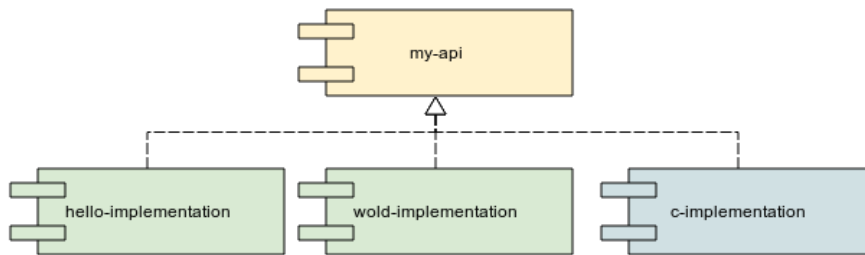
This status-overview will check for agents that have not responded for a specific time or from which error-messages were pass through.

This service (basically a json-format file) can be polled from monitoring-tools like nagios which can then make the proper communications and/or actions.

7.8 Reference implementation and specifications

In the previous drawings there were modules marked in **yellow** and other ones in **green**.

The yellow modules can be considered **specifications** (or abstract modules), they contain no concrete implementation but only data-types and interfaces to be implemented.

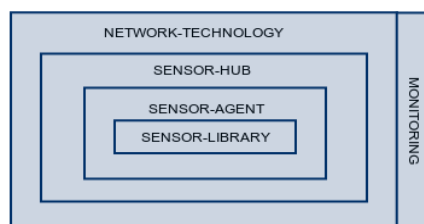


(in some case some of the implementations are in blue when we explicitly want to indicate that it concerns implementations only provided in c)

These specifications (or abstractions) allows the software to plugged in different systems and to switch from implementations when needed (e.g. migration to another type of storage).

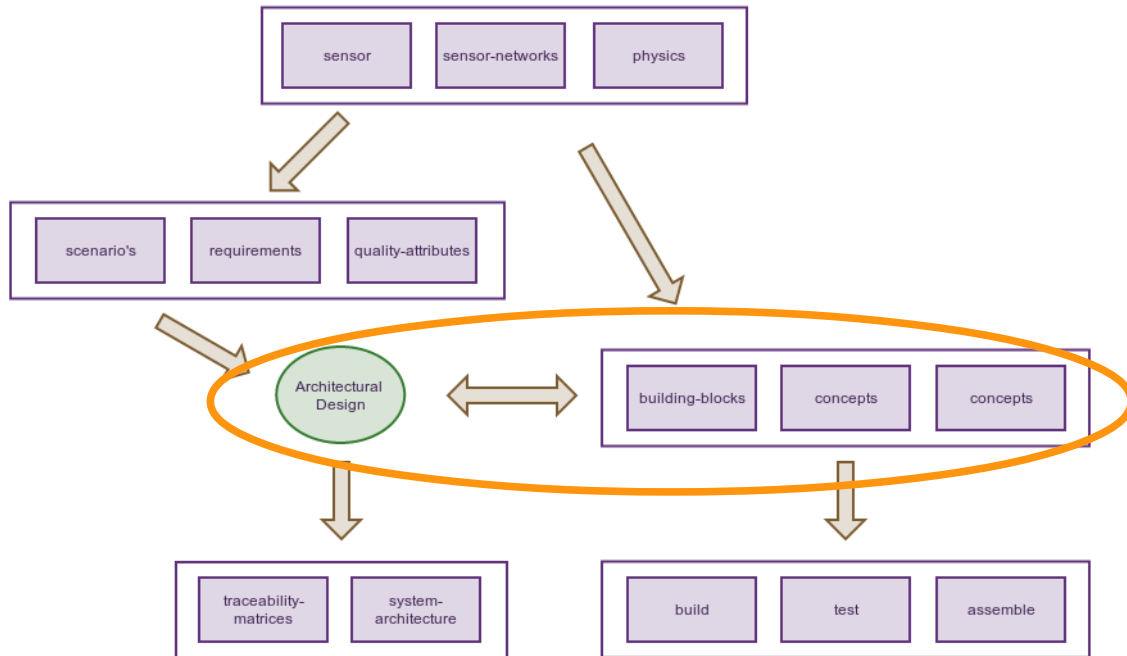
The components are in some cases also supported from different language-platforms can have different implementations.

The sensor-agent (and sensor-api) and library is e.g. supported for c-platforms (micro-controllers), the current **reference implementation** supports AVR but can be easily ported to other controllers like PIC due the necessary abstraction foreseen for the system and storage.



For Java-platform there is a support for all layers, the sensor-hub is not supported in c as this is a typical application-server-functionality; for which Java (and JEE in further extent) is specially suited.

8 Elaboration of building blocks

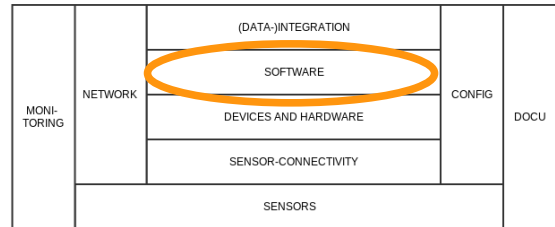


The building-blocks were defined and identified in the previous chapter

This chapter elaborates further on these building block to construct a better understanding of these building blocks and their underlying mechanisms.

8.1 Technology-stack: Languages (and platforms): C and Java

A part of the building blocks provided in for both C and Java (more specifically the sensor-api and the sensor-proxy covered in later building blocks.



Why then **duplicating functionality** over these 2 languages (and platform in case of Java)?

8.1.1 C-libraries

For the C-language there is a clear case/interest to provide these libraries:

- There is C-support for almost any micro-controller or embedded platform imaginable, C can be easily be **compiled** to **different** kind of platforms:
 - **low**-level platforms (8 and 16-bit platforms) like **AVR** and **PIC**
 - **high**-level embedded devices and computers (typical 32-bit **ARM** platforms like RaspberryPi and Beaglebone) with OS-support
- As a consequence it allows **supporti** for **constrained** environments (in case e.g. when energy-consumption is important)

C is a natural choice for micro-controllers and communication with low-level functionalities and allows us working in large-scale projects (e.g. 1000 (smart) to be monitored) where you need to take into account energy-efficiency and limited memory and storage-capabilities.

8.1.2 Java

Why should we then still support Java?

Java-applications runs only on higher devices (at least 32-bit with OS) and requires more memory than embedded other applications

- **Easy to work with:**

Java is a high-level language – when correctly developed – very

- **Portability:**

Java doesn't need to be recompiled to be supported on different platforms.

- **Library-support:**

Java is known to support complex a variety of functionalities is enormous.

Other software-building-blocks like support for database, network-connectivity, web-connectivity, GUI-applications are much more easy to be implemented (e.g. see other building blocks)

- **Server-applications:**

In other building blocks reuse this functionality to integrate in server-applications (e.g. REST-applications).

Working with Java and JEE

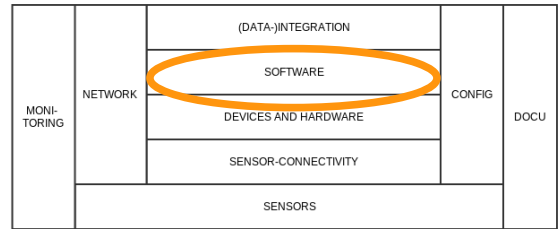
- **Language known by many**

Number of java-developers is estimated around 10 million, see the article

<http://java.dzone.com/articles/how-many-java-developers-are>

8.2 Principle applied: Modularity and dependency-injection

Before diving into the elaboration of the other building-blocks some explanation (and justification) is needed on the characteristics of the software-building-blocks.



8.2.1 Definition of modularity

Following the definition from (Knoernschild 2012) a module can be defined as:

A software module is a deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers.

In short a module allows you to isolate your code in chunks that can be reused in different context.

It allows the user of the module

- Isolation, impact and a maintainability
- Reuse in different applications
- System independence (and Hardware Abstraction Layer)
- Providing testable code by isolating functionalities

8.2.2 Dependency injection

Dependency-injection is the passing of a dependency (a service) to a dependent object (a client), rather than allowing a client to build or find the service.

For a more profound explanation go to <http://martinfowler.com/articles/injection.html>

8.2.3 Applying modularity towards platforms

8.2.3.1 Java

Modularity and dependency-injection is done by making use of the Springframework.

More details are to be found in <http://projects.spring.io/spring-framework/> and the code referenced in the deployment-part and (Ho 2012).

8.2.3.2 C

For c-development a similar style (allowing us to work modular and object oriented) is described (and followed by this project) in (Grenning 2011)

8.3 Module: Sensor-API (library)

8.3.1 Context

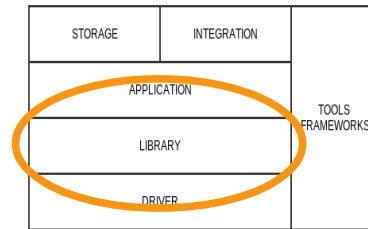
The initial demand towards this thesis/project was to deliver a library (and code) that can be used and integrated in applications for on embedded devices like :

- Raspberry Pi
- AVR (and Arduino-boards)
- Beaglebone
- PIC-micorcontroller
- ...

8.3.2 Library with API

This API (application programming interface) and library should

- Be easy to use and deploy
- Be well documented (configuration-parameters, performance, ...)
- Have a clear self-documenting API
- Hiding low-level detail like sensor-connectivity-protocols (driver-part).



8.3.3 Composition of the library

The library has been set up with modularity in mind and contains 4 layers where the modules are residing:

Sensor-API:

The module containing the interfaces (Java-interfaces and header-files for the C-libraries).

Sensor-Implementations:

The module(s) containing implementations for these interface.

In general one module is implemented for each sensor-vendor.

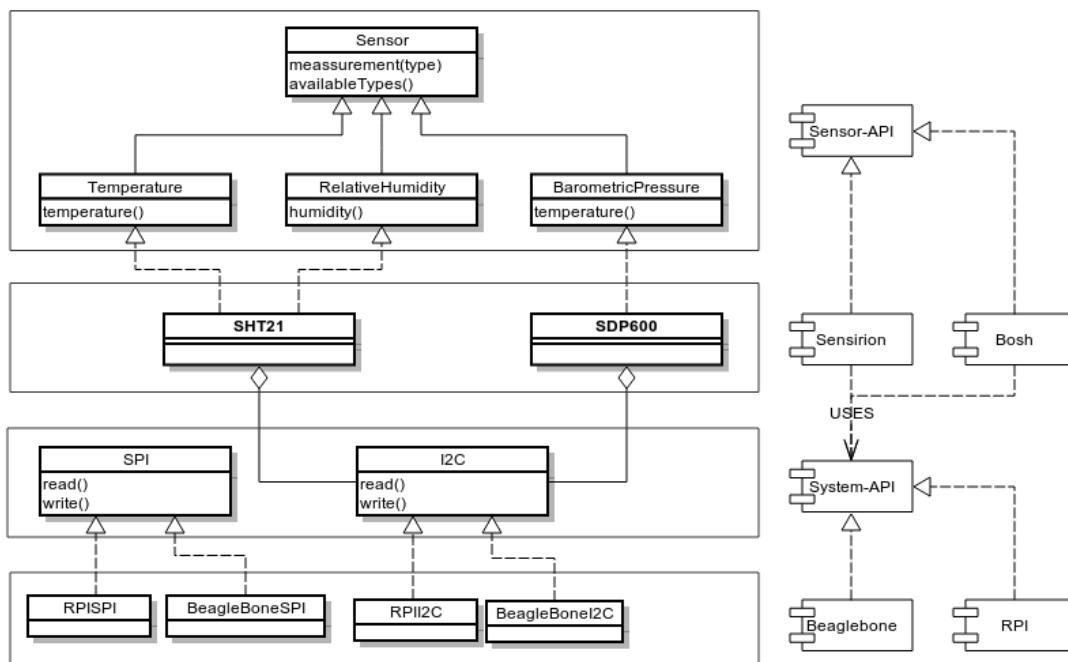
System-API:

The module containing interfaces abstracting away the system.

Sensor-Implementations:

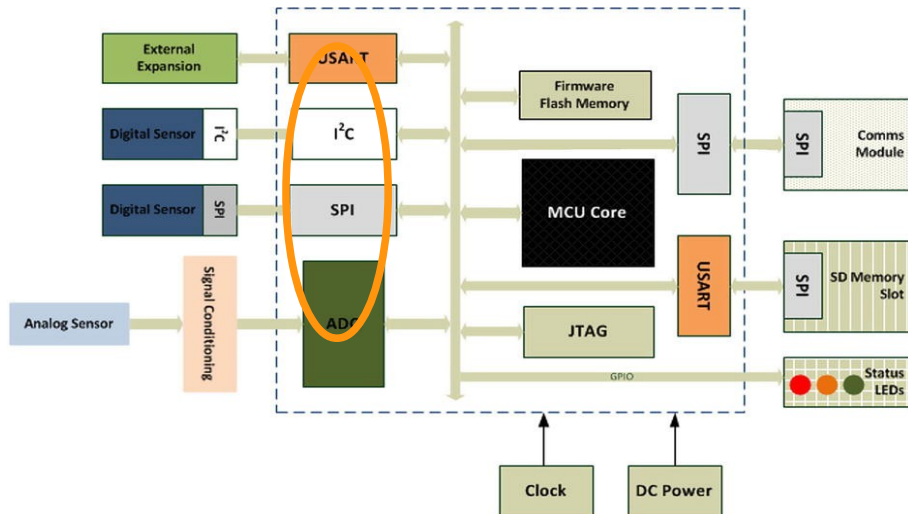
The module(s) containing implementations for the system-interfaces.

One module is implemented for each device.

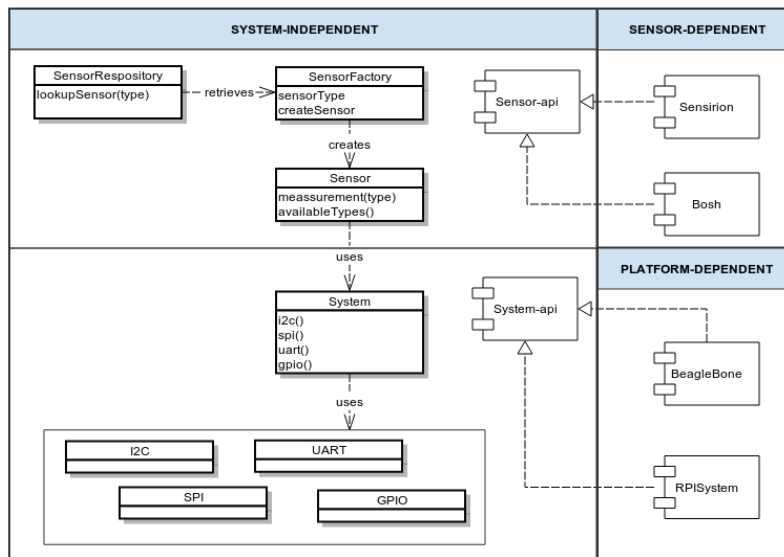


8.3.4 System-abstraction

Most modern sensors (and also the sensors used in the context of this thesis) work today with already built-in capabilities like calibration, linearisation and provide a digital interface rather than let the client analyze the signal via ADC (analog-digital-conversion).



To allow the implementation of the Sensor-interfaces to be cross-platform the Sensor-objects are injected with a System-abstraction



This allows to reuse the sensor-implementations among platforms and to make the logic inside this implementations more testable.

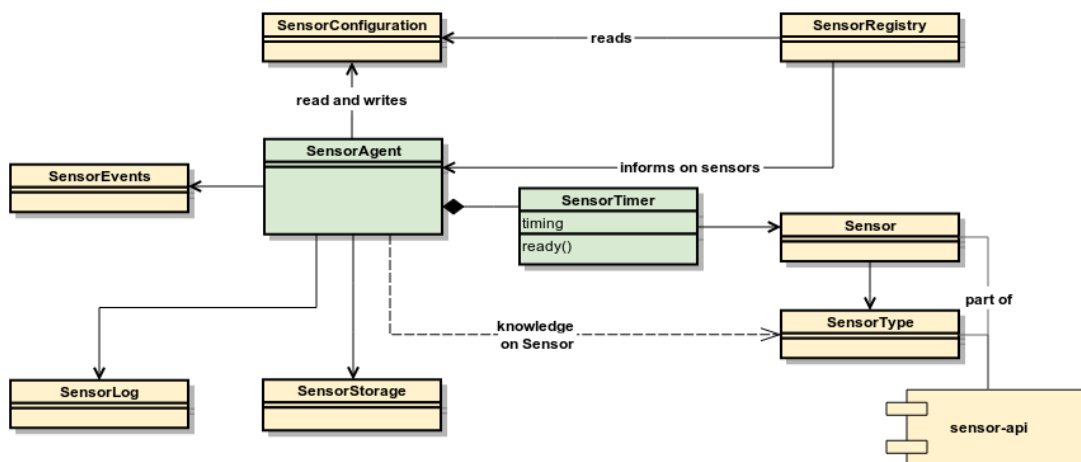
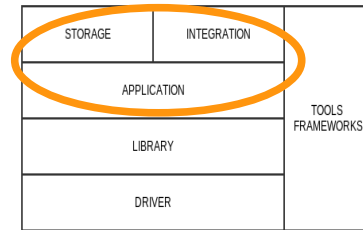
8.4 Module: Sensor-agent

8.4.1 Sensor-agent is a process

The actual execution of the agent-functionality is done by an instance of the class **Sensor-agent**.

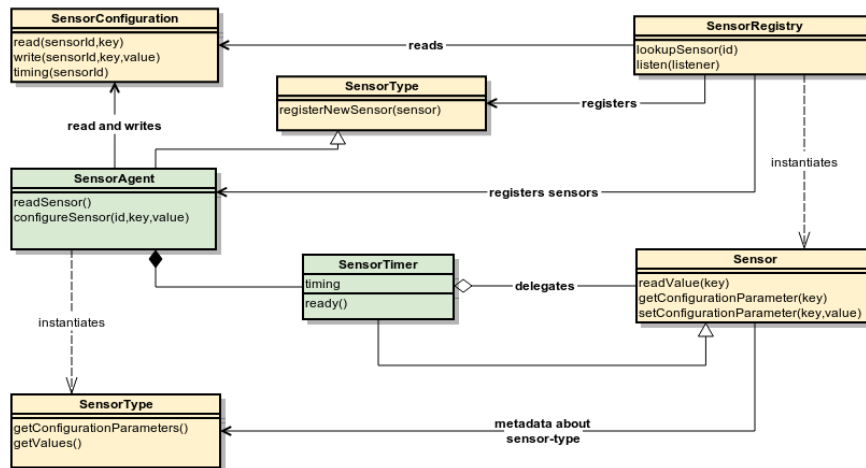
Basically this is an it's a “permanently running process” that is:

- Detecting sensors via the **SensorConfiguration**-abstraction
- Requesting data from **Sensor**-instances at periodic (configured) time-intervals while delegating to **SensorTimer**-objects.
- Handling and logging error-information (intercepted while interacting with the sensors and health-checks) and handing this information over to the **Sensor-Logger**-interface
- Writing the the data captured from these sensors to a **SensorStorage**-instance.
- Writing all events (both data-interceptions and errors) to the **SensorEvents**-interface



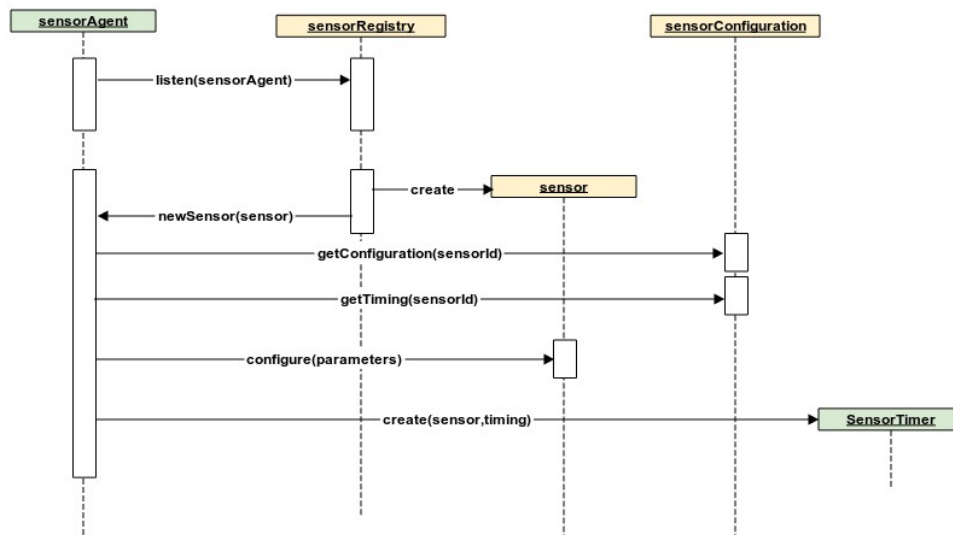
8.4.2 Starting up the SensorAgent

When the agent-process starts up the **SensorAgent** will register itself (as a listener) to a **SensorRegistry**-instance (responsible for hiding away the discovery and creation of **Sensor**-objects).



When a new sensor is registered at the **SensorAgent**, the agent will verify if the **SensorConfiguration** configuration-parameters for this specific **Sensor**.

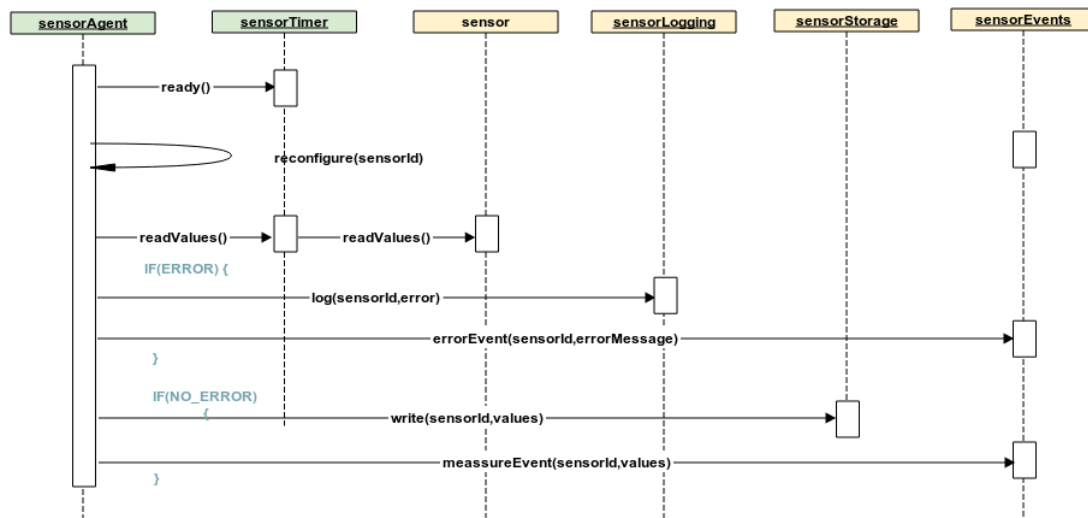
If no configuration is foreseen the **Sensor** will use it's default parameters for measuring.



After this **Sensor**-configuration, the **SensorAgent** will create a **SensorTimer** (pointing to this **Sensor**) and initialize it with a timing (default timing used for the **SensorType** if none exists in configuration).

8.4.3 Measurement-process

The SensorAgent (being the controlling process) will repeat each specified time (depending on system-configuration) a series of actions against the Sensor that are attached (via the SensorTimer-objects)



For each SensorTimer-object attached to the SensorAgent will:

- Check if the sensor is ready to measured. (the SensorTimer keeps track on the time expired since last measurement and will inform return true if this timing is expired)
- Before measuring the Sensor, the agent will check - against the SensorEvents-instance - if there are instructions present to be executed (and will reconfigure the Sensor when new a sensor-configuration is present).
- After that it will execute the measurement(s) (if the SensorTimer is ready) by delegating this to the Sensor-instances.

If **no error** was triggered from this measurement-process (Sensor) the agent will

- A measurement will be pushed towards the SensorEvents-interface
- The measured values are written to the SensorStorage (local storage)

If an **error** is triggered (from reconfiguration or measurement)

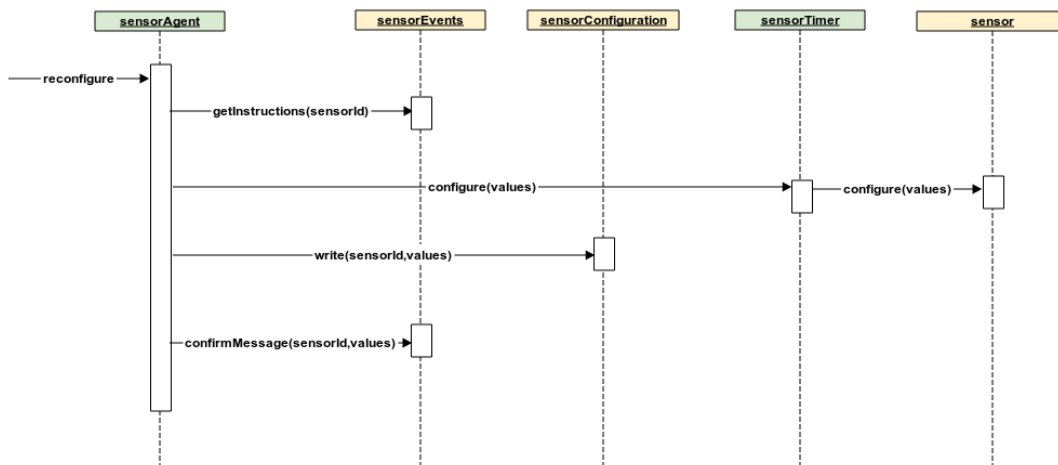
- An error-event will be pushed to the SensorEvents-interface
- The error will be (also) logged locally via the SensorLogger-interface

8.4.4 (Re)Configuring the Sensor

During the measurement-process the SensorAgent will check each time (before measuring) whether the configuration of the sensor needs to be changed.

The SensorAgent does this by checking/reading from the SensorEvents if there are new instructions (for reconfiguration) available like e.g.:

- New configuration-parameters for the sensor-agent
- Other timing required for polling the sensors



The SensorAgent will then apply these changes and - in case this is applied successfully - he will push a confirmation-message to the SensorEvents-interface.

8.4.5 Local storage

The SensorStorage-module stores the data that is intercepted by the SensorAgent from the connected sensors.

The data-model of this storage is described in the datamodel-building-block.

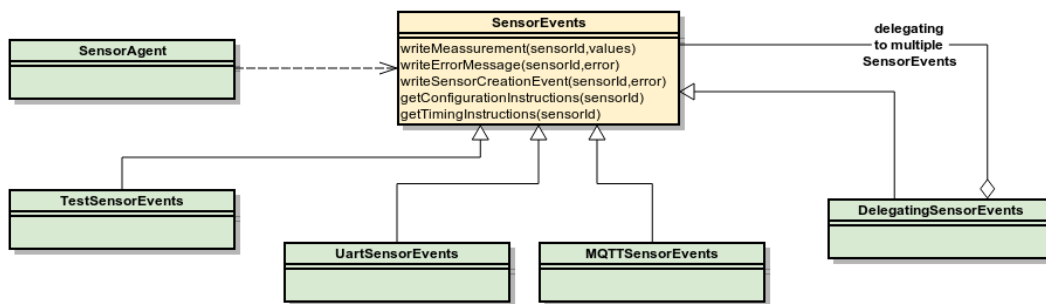
8.4.6 Integrating and events

In the flows (described in the previous pages) the SensorEvents-abstraction was used to send out messages to the outer-world like e.g.:

- Results of measurement
- Errors during messages
- Activation of sensor in a SensorAgent
- Confirmations of configuration-changes
- Evolutions in the state of the sensor

or to receive messages like e.g.

- Changing the configuration of a sensor
- Changing the timing of configuration



Different implementations of this interface are foreseen:

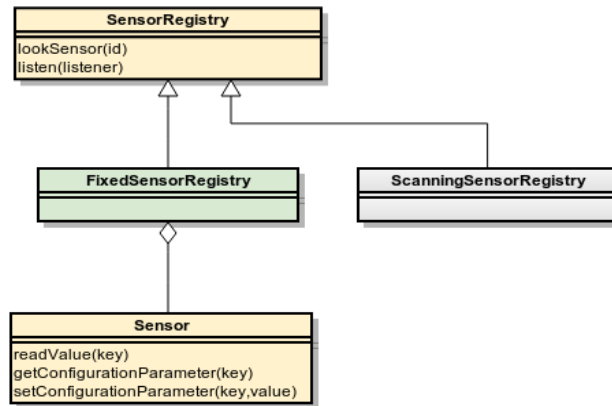
- **UartSensorEvents** will allow a client to communicate via serial communication (via e.g. an usb-to-serial interface)
- **MQTTSensorEvents** will use messaging-bus to communicate to a server
- **DelegatingSensorEvents** will delegate all messages it receives towards other injected **SensorEvents**
- The **TestSensorEvents** is foreseen for local (unit)-testing against the **Sensor-Agent**

This interface represents the communication for the **SensorAgent**, in essence it's the port toward other devices or clients interested in the output of the **SensorAgent**.

In the next chapter – when we discuss the **SensorHub** – we will built further on this communication-capability.

8.4.7 Implementations for SensorRegistry

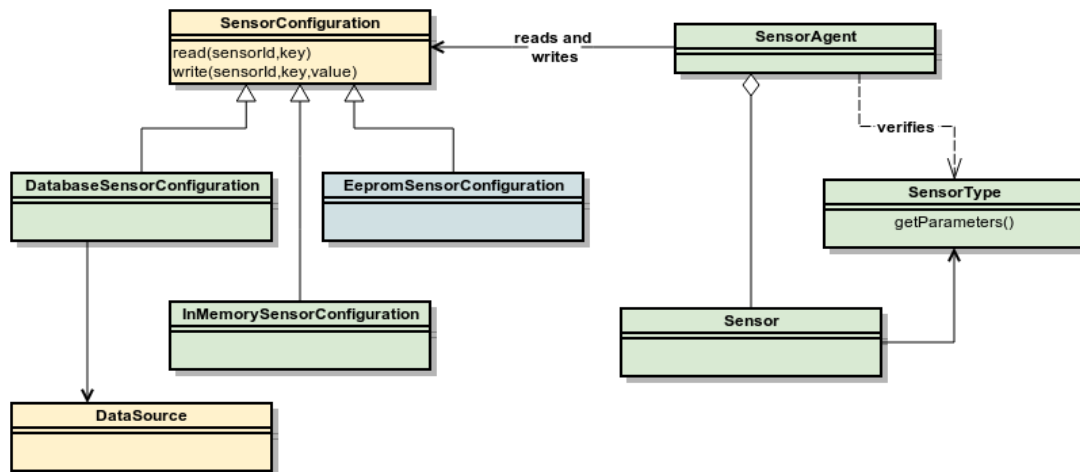
The reference-implementations(s) only foresees one implementation of SensorRegistry named **FixedSensorRegistry**.



This implementation is loaded at start-up of the application (agent-process) from a configuration-file.

A more dynamic version might be foreseen in the future, but for now we assume that - when we build and deploy - we know exactly which sensors are connected to the hardware.

8.4.8 Implementations for SensorConfiguration



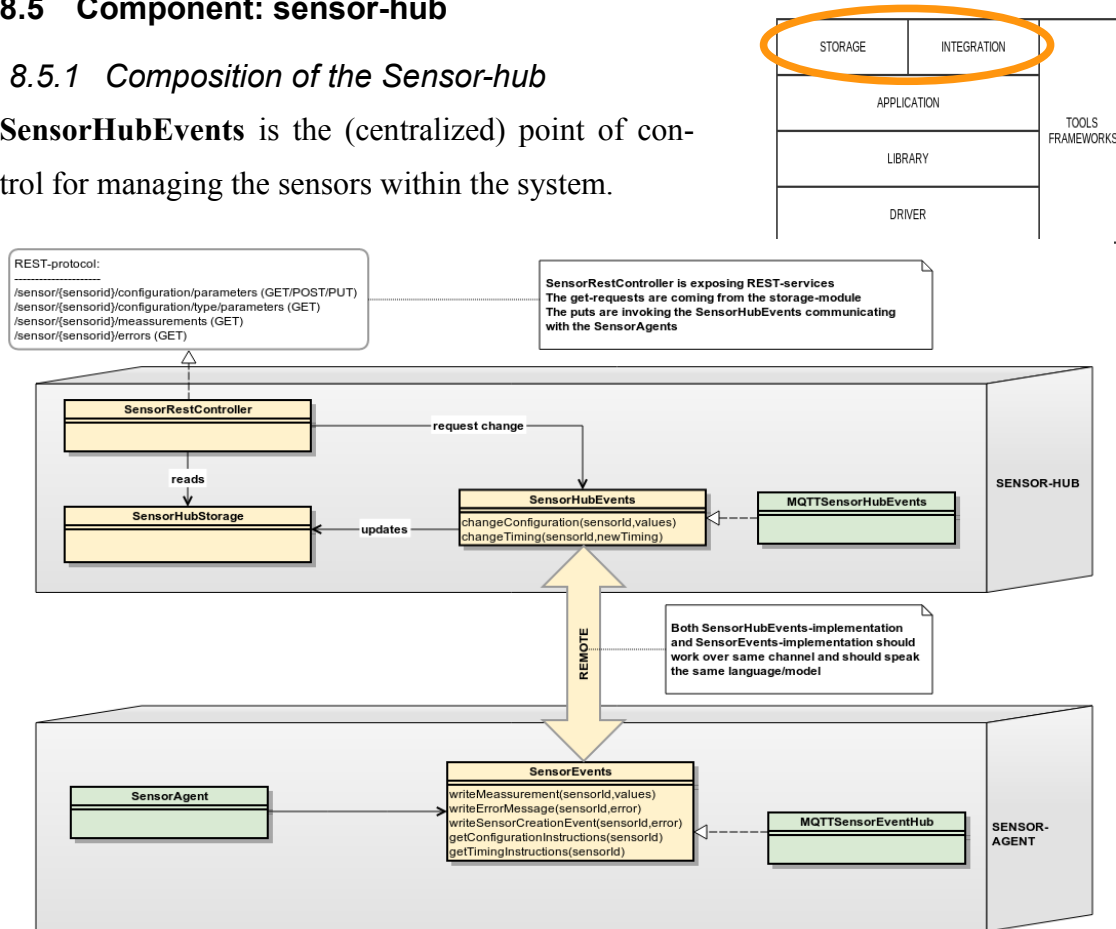
In the reference-implementations 2 versions of the SensorConfiguration-interface are foreseen:

- an in-memory (mostly to allow unit-testing testing)
- an implementation using a database (via jee-datasource-abstraction)

8.5 Component: sensor-hub

8.5.1 Composition of the Sensor-hub

SensorHubEvents is the (centralized) point of control for managing the sensors within the system.



The **SensorHubEvents** has the following responsibilities:

- **Listening** to measurement-messages coming from the SensorAgents and **storing** this to the centralized-database (more details in the data-model explained in the next building-block)
- Accepting **sensor-configuration-changes** from users of the system (via **SensorRestController**) and **routing** these changes to the **sensor-agents** (and storing error-messages if necessary)

8.5.2 Event messages

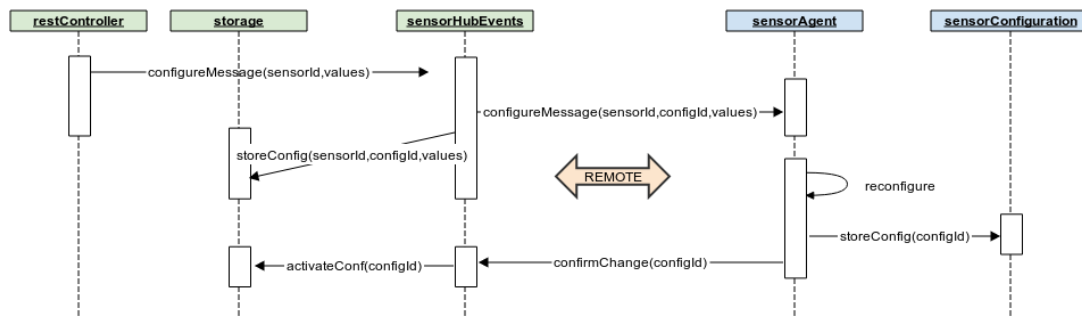
The role of the Sensor-hub can be best understood by looking at the messages-send between the **sensor-agents** on one hand and the **sensor-hub** on the other hand

8.5.2.1 New configuration pushed to the sensor-agent

When a sensor is known by the sensor-hub the client might want to change this configuration (via the REST-connector).

In this case the **SensorHubEvents**-instance is requested to configure a sensor (identified by it's sensor-id):

- The **SensorHubEvents**-instance will first create an new configurationId
- It will store this configuration (with configurationId) in the database but marked/annotated with a non-active-state

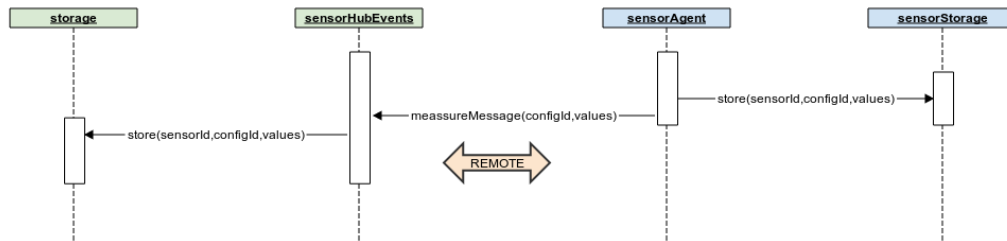


- When the **SensorAgent** receives this message (instruction) it will reconfigure the Sensor with these values and stores this new configuration in its local storage (together with this configurationId to keep traceability).
- After this the **SensorAgent** confirms this back to the **SensorHubEvents** by sending a confirmation-messages and updates the database by activating this sensor-configuration.

8.5.2.2 Measurement pushed via the sensor-agent

When the **SensorAgent** triggers a new measurement it will:

- Store the measurement-values in local storage with a reference to the configuration-id.
- Push a measurement-message to the sensor-hub

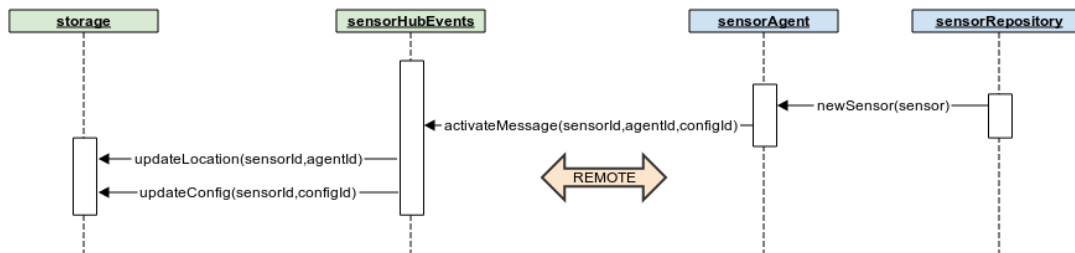


- The **SensorHubEvents**-instance will store this data (associated with the configurationId)

8.5.2.3 Activation of sensor and sensor-agents

When the **SensorAgent** comes active it has responsibility to register itself at the sensor-hub.

It does so by sending a message to the sensor-hub (the way of discovery depends on the different implementations of the SensorEvents and the SensorHubEvents-counterpart).



8.6 Data-storage

8.6.1 Local and remote storage

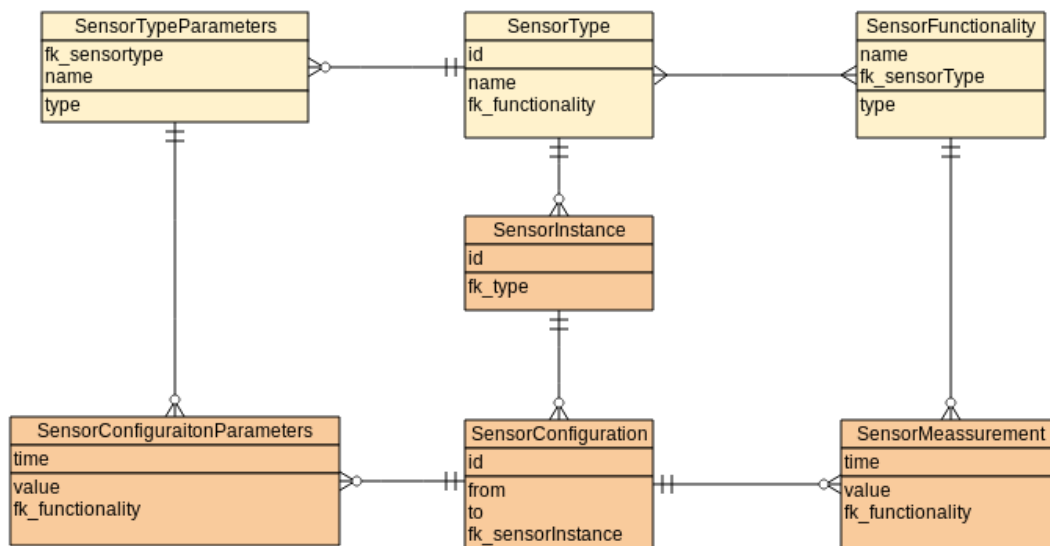
Both sensor-hub as sensor-agent are storing data (in the case the sensor-agent this is many cases to provide a backup and not to lose data in worst case scenario's).

The storage of a sensor-agent (also called local storage) is optional and can be deactivated in the configuration of the sensor-agent.

This storage can be also be rolling, which means old data can (when configured) automatically cleaned up after the maximum amount of data is reached (which makes sense in constrained environments)

8.6.2 Data-model for Sensors

The data-model used for both local storage stored in the database is the following:



The light-colored part isn't actually representing storage, this data is present in the code, more precisely it represents the metadata contained with the SensorType-implementations.

The reason of adding these (virtual) tables is that the data - stored in the database - refers to this metadata (keys parameters and functionalities of the SensorConfiguration)

PART 5: Roll out

9 Git and GitHub

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It out-classes SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

GitHub is a web-based hosting service for software development projects that use the Git revision control system.

GitHub offers both paid plans for private repositories, and **free accounts** for **open source projects**. The site was launched in 2008 by Tom Preston-Werner, Chris Wanstrath, and PJ Hyett.

GitHub is home to over 11.7 million repositories, making it the largest code host in the world.

The site provides social networking functionality such as feeds, followers, wikis (using gollum Wiki software) and the social network graph to display how developers work on their versions of a repository.

9.1 Documentation and code for this project

For this project Github provides us a centralized location to share code and documentation and allows us to keep track of changes.

All the git-repositories are to be found inside the authors github-account:

<https://github.com/bartvoet>

The main git-repository containing this thesis, appendixes, datasheets is to be found at the url:

https://github.com/bartvoet/sensors_documentation

This repository contains

- this thesis,
- the documents referred from the appendix and
- A wiki page pointing the other artefacts that were delivered in context of this project:
 - Sensor-agent (C and java) and sensor-hub (Java) implementations
 - Example projects covering different scenario's documented in this thesis (pdf-documentation, example-code, cad-diagrams ...)
 - Structured documentation on sensors
 - Database-scripts needed to setup
 - Installation guide

10 Supporting modularity in Java

In order to support modularity 2 important tools/frameworks are used

Maven is a project management tool which encompasses a project object model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle. When you use Maven, you describe your project using a well-defined project object model, Maven can then apply cross-cutting logic from a set of shared (or custom) plugins.

The **Spring Framework** provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

The **Maven model** allows us to divide our code in modules (groups of classes) and configure which module depends on which one.

The **Spring Framework** adds to this the capability to declaring and wiring objects at start-up of an application and builds on the principle of dependency injection.

Example given:

On the right you see a snapshot of the sensor-maven-project.

As an example we focus on the storage-api as it was documented in the previous part (design and architecture).

The module sensor-storage-api contains the api (interfaces not the implementation).

Beside this api you have 2 implementation:

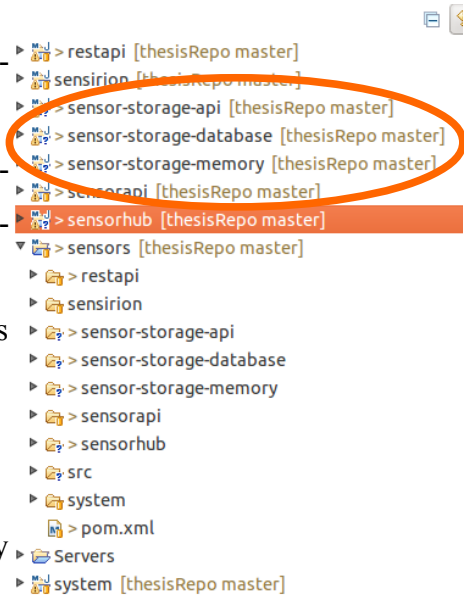
- **sensor-storage-memory**
storing the measurements in memory and typically used for testing
- **sensor-storage-database**
module delegating the storage to a jdbc-compliant database

The following snippet is an example is a snippet coming from the dependency management of the sample-setup of the sensor-hub:

```
...
<dependency>
  <groupId>org.b.v</groupId>
  <artifactId>sensor-storage-api</artifactId>
  <version>${project.version}</version>
  <scope>compile</scope>
</dependency>

<dependency>
  <groupId>org.b.v</groupId>
  <artifactId>sensor-storage-database</artifactId>
  <version>${project.version}</version>
  <scope>runtime</scope>
</dependency>
...
```

If I want to change the implementation to the database-version it's sufficient to change the artifactid to sensors-storage-memory when you package the application for deployment.



This is partly also enabled by the Spring-framework, since all storage-modules have to contain a spring-configuration-file (declarative xml) with the name `storage-context.xml` containing the instantiation of an object with the name `storageRepository`.

Configuration of the sensor-hub depends on a file `storage-context.xml` and is injected with a object of the name `sensorRepository` implementing the interface `SensorRepository` (the actual implementation depends on whether the databases or memory implementation-module is loaded:

```
<import resource="classpath:storage-context.xml"/>

<bean class="org.b.v.sensors.hub.SensorHub">
  <constructor-arg index="0" ref="sensorRepository"/>
</beans:bean>
```

Configuration of the sensor-storage-database-module

```
<bean class="org.b.v.sensors.storage.DatabaseSensorRepository">
  ...
```

For more information on this style of programming see (Knoernschild 2012)

11 Supporting modularity in C

In Java you didn't need to recompile to switch the implementations, you only needed to package together the necessary libraries (and in some cases you can even switch these capabilities at runtime)

For obtaining this kind modularity in C you can also achieve this modularity but in a slightly other way.

Imagine that you want in the C-applications apply this modularity for the configuration-capability.

You can enforce this at compile-time by defining in the system the header-file `sensor_configuration.h`:

```
...
sensor_config_t sensor_configuration_readSensorConfiguration(uint16_t id);
void sensor_configuration_writeSensorConfiguration(sensor_config_t config);
...
```

and importing in the C-implementation of the sensor-agent this header-file

```
...
#include "sensor_configuration.h"

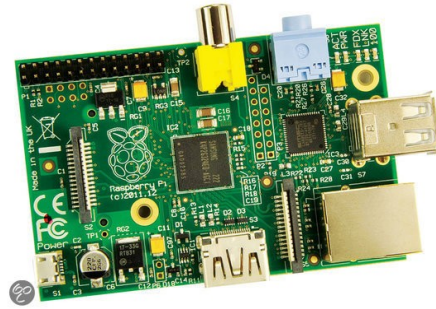
int main(void) {
    setup();
    for (;;) {
        sensor_storage_readSensorConfiguration();
        ...
    }
    return 0;
}
...
```

Depending on which implementation you include you can change the behaviour or the storage as you could do for the Java-part (but at compile-time).

12 Processing devices

12.1 Set up with Raspberry Pi

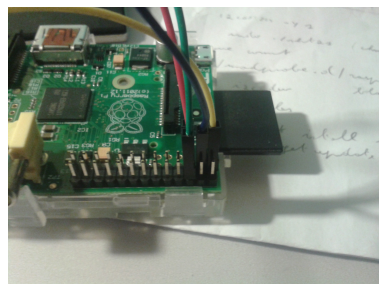
For the development of the prototype of the sensor-hub we used a Raspberry Pi with Java version 7 (although the compilation will now migrate to Java 8).



For this test we connected a **sht21**-sensor (Sensirion – Relative Humidity) to one Raspberry Pi and a **sdp600** (Sensirion – differential pressure) to another Raspberry Pi.

Both sensors work on 3,3 v which is the same voltage-level as used for the Raspberry Pi, no logic converter was required.

Since the RPi board already has 1.8K resistors on the I2C lines, any pull-ups included on a breakout board are superfluous so the the sensors could be connected directly.



For the communication both devices were connected to the same router and the messaging-product used was ActiveMQ (activemq.apache.org).

12.2 Arduino Micro Pro 3,3 v

For the development of the prototype for the micro-controller-targets we used and Arduino Micro Pro:



The normal Arduino work with 5 v, in order to avoid the usage of a logic-level-converter we selected this version which is working with 3,3 v on 8 mHz rather than 16mHz.

This prototype required us the usage of an 4k7 pull-up-resistor

Although we used an Arduino the code was written in C and not with the arduino-language-platform.

We want to make this code efficient and more portable to other AVR-controller and even other architectures (see also (Williams 2014b) for more coverage on this).

13 Interacting with the application (REST-full services)

This is more detailed described in the system-documentation (to be found in appendices) but this should give an idea (an overview) of you can work with the system..

13.1 What is REST?

REST stands for Representational State Transfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, communications protocol -- and in virtually all cases, the HTTP protocol is used.

REST is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

The advantage of using this rest-full services is that this can be easily integrated for both server-to-server-integration as for mobile applications as for web-applications.

If your interested in this topic, for more information on REST-full-architectures check (Saternos 2014) or (Jim Webber & Robinson 2010)

13.2 What is JSON

The serialization-format supports is JSON.

JSON, or JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

It is used primarily to transmit data as an alternative to XML.

13.3 List of available REST-full services

List of sensors connected to the system: (return json with id, location, ...)

GET `http://{server}/sensor_connector/sensors/`

For a specific sensor the measurements that are supported:

GET `http://{server}/sensor_connector/sensors/{id}/types`

If you want to know what configuration-parameters are supported:

GET `http://{server}/sensor_connector/types/{type}/parameters`

If you want to configure the configuration-values for a specific sensor:

POST `http://{server}/sensor_connector/sensors/{id}/types/{type}/parameters`

If you want to consult all the measurements:

GET `http://{server}/sensor_connector/sensors/allmeasurements`

GET `http://{server}/sensor_connector/sensors/allmeasurements?from={time_from}&to={time_to}`

If you want to consult on a specific sensor:

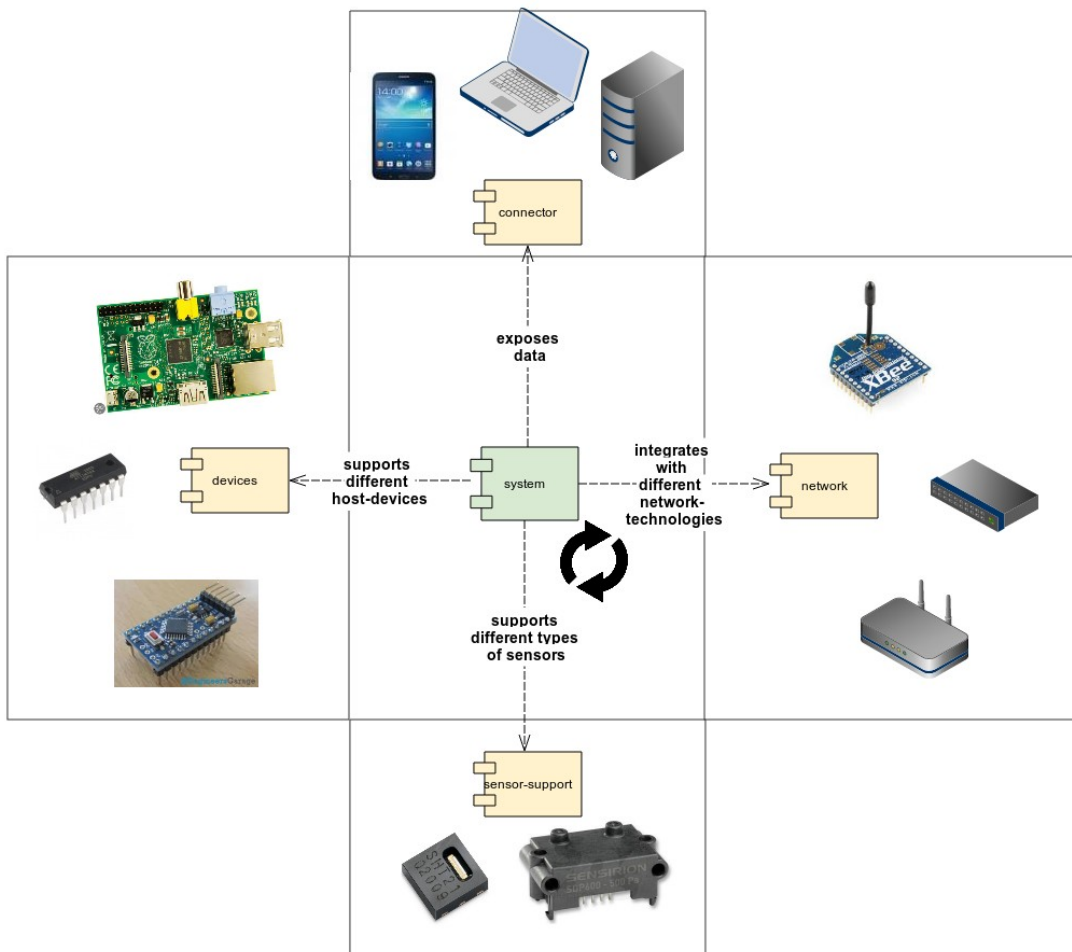
GET `http://{server}/sensor_connector/sensors/{id}/measurements`

GET `http://{server}/sensor_connector/sensors/{id}/measurements?from={time_from}&to={time_to}`

PART 6 : Conclusion

14Conclusion

The drawing below was used at the start of thesis to demarcate the scope of this project/thesis.



14.1 Sensor-support

Support for the required sensors (sht21 and sdp600) is to be provided for 2 platform/languages; Java and C.

14.2 Support for multiple devices

The C-implementation allows us to run code on any platform by providing an abstraction layer for system-dependent-parts.

The Java-implementation gives us the same capabilities but it allows to integrate this in java-applications which are today very popular on embedded platforms (Beaglebone, Raspberry Pi)

The Java-part is fully implemented (sensor-libraries, sensor-agent, sensor-hub)

The C-part is partially implemented (sensor-libraries), the sensor-agent capabilities is under construction.

14.3 Network-Integration

An abstraction is foreseen for the communication between the sensor-agent and the sensor-hub.

This current implementation is based on ActiveMQ but other implementations can be built in the future with affecting the other modules.

14.4 Client-Integration

The client can connect to the application requiring nothing more then a browser (or optionally the use of the curl-command).

Via a uniform (REST-)interface the client can change configuration, consult measurements, monitor errors on the system

15Appendices

Appendices referred from this thesis are to be found at

https://github.com/bartvoet/sensors_documentation/thesis/appendices

16Bibliography

Primary sources

- Bass, L., Clements, P. & Kazman, R., 2013. *Software Architecture in Practice, Third Edition (bart voet's Library)*, Addison-Wesley Professional.
- Bell, C., 2013. *Beginning Sensor Networks with Arduino and Raspberry Pi*, Apress.
- Bently, J.P., 1995. *Principles of Measurements Systems*,
- Catsoulis, J., 2009. *Designing Embedded Hardware*, O'Reilly Media.
- Fraden, J., 2010. *Handbook of Modern Sensors*, Available at:
<http://www.springerlink.com/index/10.1007/978-1-4419-6466-3>.
- Grenning, J.W., 2011. *Test-Driven Development for Embedded C (for Bart Voet)*, The Pragmatic Bookshelf, LLC (422319).
- Ho, C., 2012. *Pro Spring 3*, Apress®.
- Jim Webber, S.P. & Robinson, I., 2010. *REST in Practice*, O'Reilly Media.
- Knoernschild, K., 2012. *Java Application Architecture: Modularity Patterns with Examples Using OSGi (bart voet's Library)*, Prentice Hall.
- Martin, R.C., 2009. *Clean Code (Bart Voet's Library)*, Prentice Hall.
- Matt Stephens, D.R., 2010. *Design Driven Testing Test Smarter, Not Harder*, Apress®.
- McGrath, M.J. & Scanail, C.N., 2014. *Sensor Technologies*, Apress.
- Rozanski, N. & Woods, E., 2012. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives (Bart Voet's Library)*, Sams.
- Saternos, C., 2014. *Client-Server Web Apps with JavaScript and Java*, O'Reilly Media, Inc.
- Tero Karvinen, K.K. & Valtokari, V., 2014. *Make: Sensors*, Maker Media, Inc.
- Williams, E., 2014a. *Make: AVR Programming*, Maker Media, Inc.
- Williams, E., 2014b. *Make: AVR Programming*, Maker Media, Inc.
- Wilson, J.S., 2004. *Sensor Technology Handbook*.