NTNU – Trondheim
Norwegian University of
Science and Technology

Flexible and Scalable Local Controller for a Micro Grid Home in the e-GOTHAM Project

Thaddäus Hausler

Embedded Computing Systems
Submission date: July 2015
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Thesis

A Smart Grid Application

NTNU – Trondheim
Norwegian University of
Science and Technology

Flexible and Scalable Local Controller for a Micro Grid Home in the e-GOTHAM Project

---

Author:
Thaddäus Hausler

Supervisor:
Professor Sverre Hendseth

July 8, 2015

Contents

1 Abstract

Recent trends towards a smart grid implementation pursuits the reduction of consumption peaks, amongst other effects. A flexible and scalable Micro Grid solution breaks this problem into parts by localizing the solutions. The control over the reaction to consumption peaks is partly shifted towards the local Controller.
This report introduces a design of a localized home Controller within the e-GOTHAM framework. It attempts to construct a flexible, scalable and customizable Controller that uses consumption data and different incentives to schedule Utilities.
The proposed design splits the specified requirements into a direct controller for each Utility and an optimisation section. The optimisation creates a ranking for utilities using the provided incentives. The two parts are connected via few straight forward communication methods. The division of functionalities ensures flexibility in the implementation of optimisation algorithms and Utility controls, as well as scalability in the amount of Utility controls and algorithms used.
A proof of concept is introduced which provides uncomplicated instructions on the implementation of different versions of this design. As a result, the Controller can be used within an e-GOTHAM context with little adaptations and even in systems providing similar data input as the e-GOTHAM box.

List of Figures

List of Tables

List of Listings

2 Introduction

2.1 Motivation

On the 20th of March 2015 a solar eclipse took place and rendered vast quantities of photovoltaic energy sources (PV) all over Europe powerless. In the 2.5 hour event the generation of energy dropped by 15 GW in the case of Germany, which has the highest amount of PV in Europe.
The event was anticipated, so months of planning were able to prevent a major grid failure. It enhances the case for the implementation of Smart Grids to enable higher flexibility in load control and reaction time.[13]
Smart Grids are an attempt to modernize the power grid to enable better control and more flexibility with the trend of generating power through renewable, green energy. An incentive is given by the call of the European Renewables Directive where countries promised a certain percentage of renewable energy usage. Norway promised to set the amount of renewable energy consumption to 67% by 2020.[1]
In the case of Norway they invest a lot into Smart Grids. An example is the desire to implement smart meters by 1st of Jan. 2019 [20].
The requirements for balancing the load in the Norwegian power system are excellent. Not only is the main supply of energy provided by 98-99% hydro power, but the consumption can be highly flexible as well. Electrical vehicles are subsidized by the state and heating in houses is mainly provided by electrical heaters. The widely distributed broadband connections between major parts of the country reduce the investment into this type of infrastructure.
On the other side of the equation, the Norwegian grid is weak in parts, which strengthens the case for Smart Grid development.[9]
The e-GOTHAM project is a European cooperation project dedicated to the development of smart, flexible and scalable Micro Grids. To this purpose e-GOTHAM is developing a distributed embedded system to meet the need of gathering information and localized control of consumption. This control is mainly directed at reducing power spikes from the source to take stress off the grid, enabled by incentives like an electricity price with two tiers and information about the price in the immediate future.[2]

2.2 Previous Work

On the subject of controlling consumption in a local home, many papers and theses have been written. In direct regard to e-GOTHAM, only an internal attempt at implementing an other form of scheduler using Matlab has been made to the knowledge of the author. Other than the mentioned internal proposal, no other work exists to fill the specifications of this thesis.
Demand Side Management programs (DSM) have been proposed since the late 1970s and used to incentive consumers during utilization of electricity in order to shape the utility load curve. DSM can enable two way communication between consumer and grid, to schedule house hold appliances and coordinate between the two actors. Heavy loads during peak hours can be scheduled to off peak hours[17]. DSM is capable of managing available generating capacity without installing new power generating infrastructure [8].
Incentives for scheduling the appliances is discussed in [11], while direct load control is discussed in [22] and [24].
An interesting approach is done in [12] where Utilities are scheduled in run time. Different heuristic values are assigned to the Utilities to determine priorities.
[16] classifies home appliances after a task model. The control of the schedule is done by considering different factors like operation length, deadline, actuation time and a consumption

profile.
Game theory is applied in [18]. Users are the players and their daily usage of Utilities are classified as strategies. The schedule is reduced to a minimization problem which in turn reduces the peak load.
An attempt to have a real-time approach to a smart home energy system is shown in [23]. The article investigates a demand response mechanism which uses multiple different household appliances. It employs a rolling optimisation half-hour-ahead and uses fuzzy logic as controller.

2.3 Purpose and Objectives

The work described by this report relates to the previously mentioned DSM programs. The objective is to design a system, that uses incentives like the future prices and a consumption cap to schedule Utilities used in local homes by the consumer during run-time.
Utilities, in the case of this report, are appliances like electrical space heaters (Heaters) and hot water Boilers (Boilers). The term "Utilities" can also be applied to other electrical appliances as well like electrical car chargers and freezers.
The purpose of this design and proof of concept is to provide a way to use incentives in an flexible way to schedule Utility usage and enable other ways to optimize a schedule. A successful optimisation is performed when the consumption of a home is reduced, while the comfort level is kept or is higher than before.
Another aim of this work is to enable scalability in the choice of Utilities. The number of Utilities utilizing the incentives can vary between sites and is therefore to be variable.
A home, used as a living space by families or other living arrangements, is the aimed at target to be optimised by this Controller. This requires a level of personalization that is built in by design.
A proof of concept is to be implemented for the design and run on an e-GOTHAM box. The e-GOTHAM box is an embedded system that incidentally records consumption and specific sensor data from the local home. This data is to be used in the Controller to make predictions about behavior in consumption for Utilities and the environment they run in.
The main purpose of this work is to provide a flexible and scalable design, that can implement new incentives for scheduling and manages the Utilities in a effective way. The proof of concept shows that the design can be adapted to the circumstances of the environment and is able to include other sources of data as well. These features are of benefit in the fast developing circumstances of smart and Micro Grids. They provide an uncomplicated way to manage the challenge of fast shifting stress on the grids and new energy solutions at a basic level and is still able to be influenced by high level decisions affecting the whole grid.

2.4 Road Map to the Thesis

This report is following the recommended structure of a thesis report, which is adapted to the needs and specific qualities of the topic.
The abstract and introduction chapters are essential parts of the report. They represent an overview of the project and the context in which it is written. The motivation that led to the problems solved with the current results and how it fits into work that has been done by others is also part of the Introduction.
The Background chapter will give the reader an overview over a broad range of topics which relate to the project itself. Information about Smart Grids, Micro Grids and on how e-GOTHAM ties into it is represented, as well as some of the tools and the concepts that are used in this thesis.
The specification, taken from the official project description, is analyzed to provide insight

into the problems, in chapter four. Specifically the Architectural design considerations and the proof of concept are identified and specified.
The Design and Proposed Solution chapter is leading the reader through the design by specifying the relationship between Entities, Actors, etc. The chapter starts with a description in broad strokes of how the design was created and why some decisions have been made. This chapter represents one major part of the solution to the questions asked in the specification.
While the design and proposed solution chapter is relating the decisions and constructs used, the Implementation chapter shows the Java implementation of the Design. All previously defined parts of the design are mentioned and show listings of code where appropriate.
The testing chapter provides not only all test cases and the resulting data, but also relates on how to implement a proof of concept with the example of the system used to test the design and functionality on the simulated e-GOTHAM box. The testing chapter is answering more of the questions proposed by the specification and gives a proof, that the design and implementation are valid.
The discussion chapter relates the expectations to the results and does an interpretation on the test data. The discussion chapter attempts to connect the controller to the e-GOTHAM project and elaborates on the limitations of the results.
Chapter number nine is the conclusion, where a short summary of the project, with the major results highlighted and a short personal opinion on the subject can be found.
The further work chapter shows ideas and give food for thought on future work in relation to the project. This includes ideas for new controllers as well as efficiency, accuracy and speed improvements for the project, that were out of scope for this work.
Finally in the last few pages, the appendix shows code and data examples that were too big for the sections they were mentioned in. They fill the last pages of this report.

3 Background

This section provides an overview over the background, the thesis is built upon. It contains information about a range of topics that are of concern when reading the remnant of the report.

3.1 Smart Grids

The term Smart Grid is used as a supplement for a range of new technological solutions for the electricity Grid of the future.
The main thesis about Smart Grid is that it gathers and analyses data from all parts of the Grid, in order to react in an efficient manner to disturbances in the electrical supply.
This data and reactions span the whole way from the generation of electricity, controlling the distribution to the consumption.
It is considered the replacement or the natural evolution of the century old power Grid. Countries like the USA rely on their power Grids, but are suspect to power outages and disturbances because smaller parts of the Grid experience failure. These outages are blamed on a lack of automated analytics and slow response times of switches[14].

3.1.1 Micro Grids

A Micro Grid is a small-scale power Grid that can operate independently or in conjunction with the area's main electrical Grid. It is usually part of a Smart Grid.
A grid qualifies as Micro Grid when it provides its own power generation, resources, loads and if it has definable boundaries. A Micro Grid can act as back up and buffer in outages or high demand periods affecting the main Grid.
The power sources of Micro Grids are often renewable solutions, like PV panels, that can be applied locally. Investments into Micro Grids can be used to reduce cost and boost reliability. The separation from the main Grid is of benefit as it can protect the area from localized disasters and make upgrades on the Grid less investment heavy, as it can be applied to one Micro Grid after the other. The practice of using Micro Grids is known as distributed, dispersed, decentralized, district or embedded energy generation[7].

3.1.2 Smart Grid in Norway

Norway faces a very individual challenge with the implementation of Smart Grid features. As mentioned in [3] the implementation of smart meters in homes is required by 1st of January 2019. The state of Norway embraces the worldwide shift towards green energy by directing parts if the investment of the state fund into renewable energy[21]. This notion is augmented by the tax exemption on electrical cars.
Multiple institutions like the Norwegian Smart Grid Center, the National Grid Laboratory and SINTEF as well as universities like NTNU are dedicating research into this topic. Various investments into the exploration of this topic is done by the Norwegian power industry, by creating areas like Demo Steinkjer, where new technologies can be tested in so called "living labs".
The Norwegian situation which the Smart Grid is based on can be described as is unique[20]. Some of the factors are highlighted as follows:

- Flexible consumption in local homes, where space and water heaters are mostly electric.
- The use of purely battery based electric vehicles in on the rise, which adds to the flexibility in consumption.

- Most electricity is generated by using hydro power plants which inherently have the ability to provide flexibility in storage and reaction time.
- Parts of the Grids are weak compared to other countries [9] and require good load balancing and investment into replacement.
- Broadband communication is wide spread and can be used for communication within the Grid.
- The multi-national energy markets[4] enable a vast quantity of load balancing options.

These factors combined provide good arguments for the step towards a Smart Grid.

3.2 What is e-Gotham

e-GOTHAM is the name of an European cooperation project with 17 European companies and universities participating. The funding is provided in Spain by the Artemis JU and the Ministry of Industry, Energy and Tourism. Its structure is within the European Artemis programs framework.

3.2.1 Concepts

The underlying concept of e-GOTHAM can be explained as a divide and conquer strategy. The overall power Grid is divided into smaller Micro Grids which contain "distributed generators, loads, storage components, a Micro Grid central controller, local controllers and a coupling point to the utility grid" [2].
e-GOTHAM aims to create the micro Grid as Smart, flexible and scalable. The developed Information system and architecture is to be open to allow third party developers to create new devices and software.
Figure 3.1 shows a conceptual representation of the micro Grid concept in e-GOTHAM, taken from their website [2]. It shows the division of smaller types of Grids from the main Grid that employs some generation and consumption of energy. The Micro Grids can be controlled individually, while the controllers also get input from the superordinate Grid.

3.2.2 Objectives

The objectives of e-GOTHAM are to implement an aggregated Energy Demand Model. The objectives can be listed as:

- Effective integration of renewable energy sources
- A dynamic matching demand and supply to increase management efficiency
- Carbon emission reduction by prioritizing green energy sources
- Monitoring products enable a higher awareness of energy consumption
- New business models for a better development of a market for energy efficient technologies.

[2]

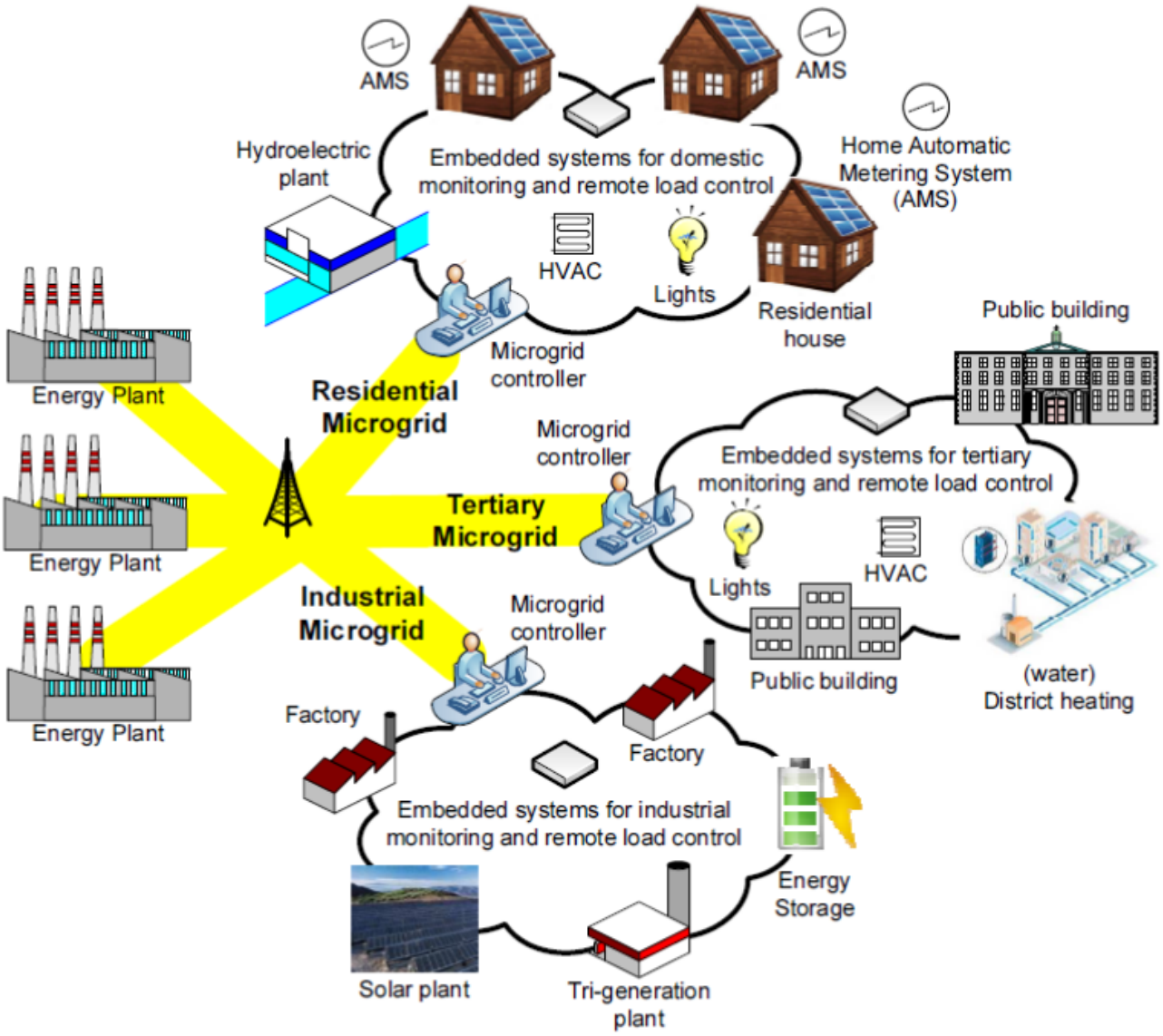

Figure 3.1: Illustration of the Concept of Micro Grids in the Context of e-GOTHAM

3.2.3 Demo Steinkjer

Steinkjer is a small community north of Trondheim.Interested parties can test smart meters and other Smart Grid related developments here. It has around 800 end users, consisting out of normal households, industry and commercial companies.
The main focus in this demo is the flexibility of end users value and added services for the distribution companies. A relevant incentive tested here is the tariff, that is used in this project as well. Steinkjer is the testing site for the e-GOTHAM boxes, developed by SINTEF.

3.3 Average Daily Usage of Energy in a Residential Housing

The point in time, when the electricity consumption is the highest during the year is called the power Grids peak demand or peak load. The highest peak demand measured in Norway until 2012 was a value of 23,994 megawatts for one hour in 2010[10].
When regarding a twenty four hour period the peak demand is found between 8 and 10 o'clock. This is due to hot water Boilers attempting to top up the hot water for showering in the morning before work. For commercial buildings this is also the time when electrical appliances are started to prepare for the work day. Figure 3.2 taken from [5] shows average usage of

electricity per hour. The graphs show a valley between the hours 1-4 in both curves. From 6-8 the buildup for the peak in household consumption is visible, while the commercial consumption starts its peak later.
While the commercial building consumption slowly declines after the peak at around 9 o'clock, the Households fall off during the lunch hours, only to peak again from around 17 o'clock. The peak of the day is reached at around 19 o'clock, after which the consumption falls off to the night level.

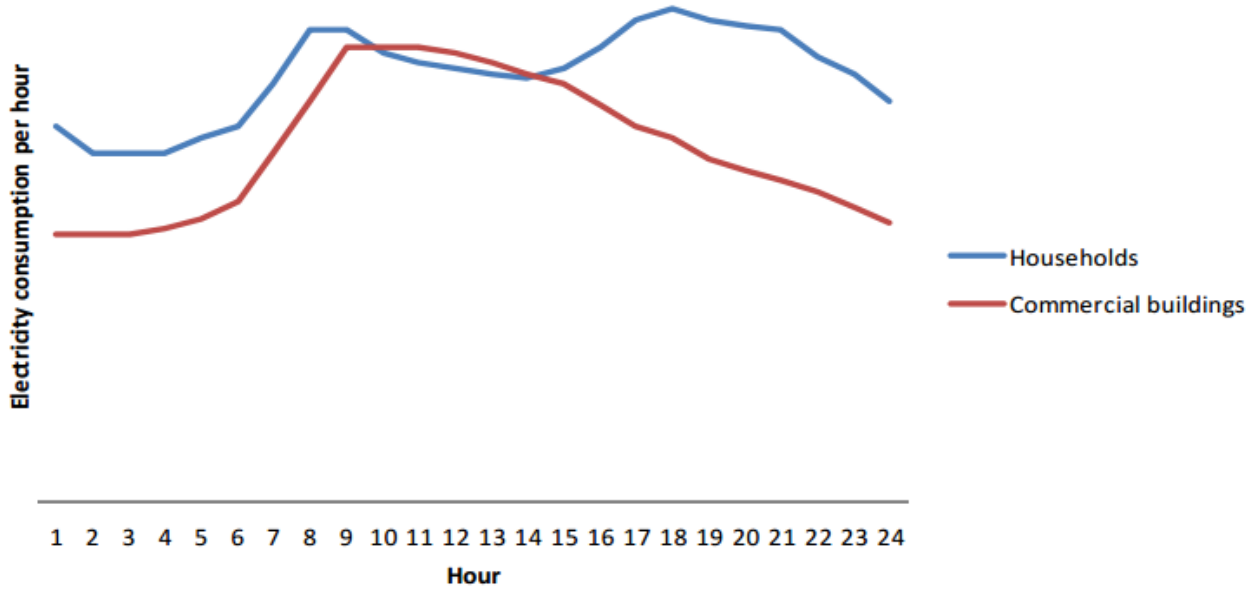

Figure 3.2: Illustration of the Daily Profile of Electricity Consumption in Households and Commercial Buildings

3.4 Raspberry Pi

A Raspberry Pi is a micro controller developed by the Raspberry Pi foundation in the UK to the purpose of teaching basic computer science in school.
A licenced manufacturing agreement was set up with companies like Newark element14, RS Components and Egoman.
The models available are model A, B and since early 2015 Raspberry Pi 2. model A and B use the Broadcom BCM2835 System on a Chip (SoC) which includes a 700MHz ARM1176JZF-S CPU with a Broadcom VideoCore IV GPU, and 256MB of SDRAM. The difference between Model A and B is the missing USB hub and Ethernet connection in model A. The Rapsberry Pi has many different ports, from video and audio out to an micro SD slot.
The operating systems provided reach from a Debian Linux, Arch Linux and an operating system called QtonPi to a number of ports that have been created by the community.
The community is what drives the Raspberry Pi to its popularity. Many different programs have been ported to the Raspberry Pi i.e. the game Minecraft and a variety of operating systems.[19]

3.5 JSON

The abbrevation JSON stands for JavaScript Object Notation. Even though it was based on the JavaScript Programming language, 3rd edition from 1999, it is a light weight data exchange format for multiple plattforms.
Because JSON works with strings as basic variable, it can be parsed by most programming languages.
The simplicity stems also from the two possible structures: A collection of name/value pairs or an ordered list of values.
Figure 3.3, taken from [6] shows the basic structure of JSON strings. An object pairs a string with a value, as seen in the first part. Objects can be lined up, separated by a comma.
An array saves only values and lines them up, separated by commas.
The key idea is, that values can be all kinds of data types, from string to numbers to an object or array. This enables an infinite possibility of arrangements, which are still parseable by human and machine.

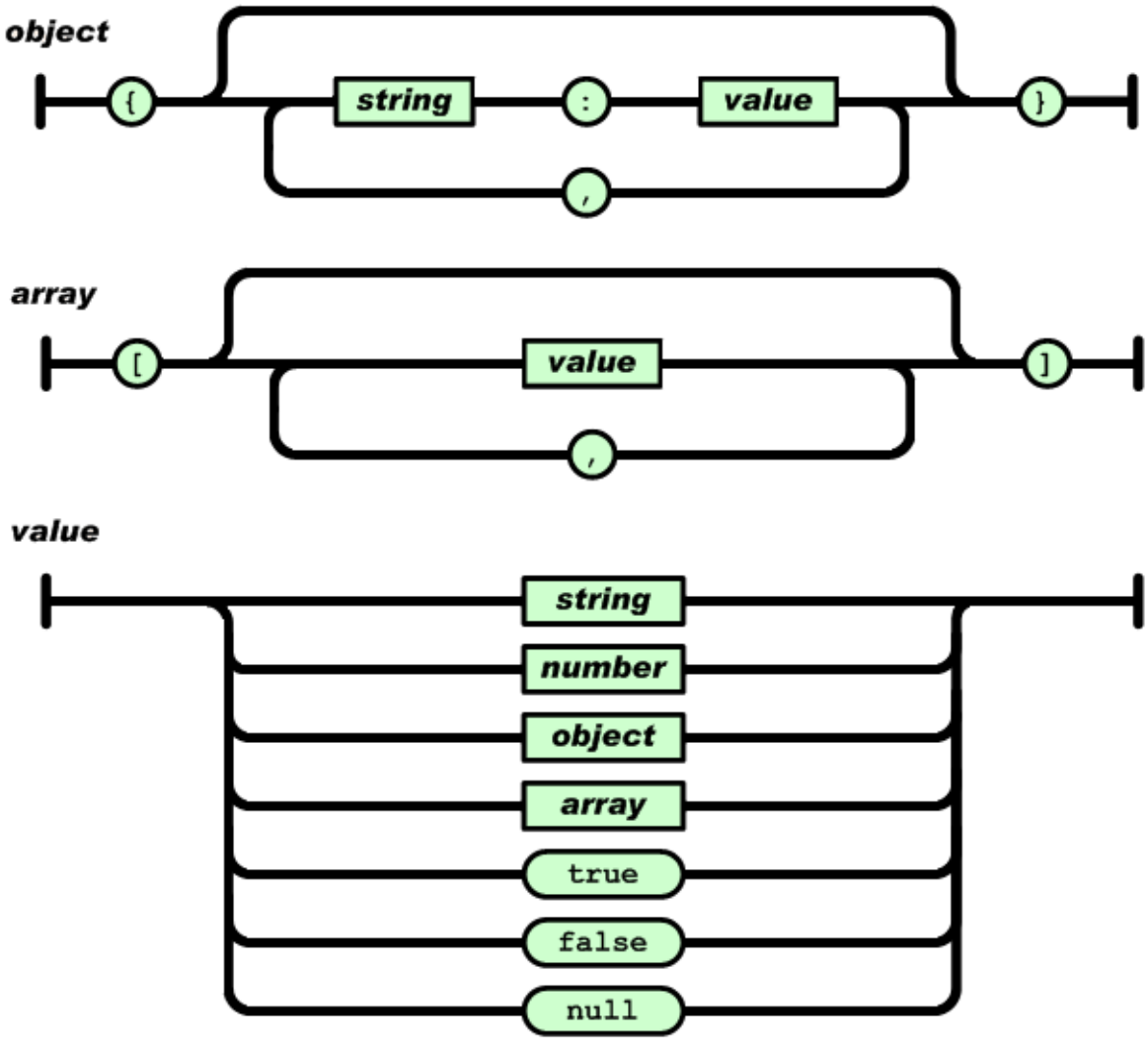

Figure 3.3: Structure of JSON Strings

4 Specifications

The specification is one of the integral parts of the Thesis and was created in cooperation between Geir Mathisen, Sverre Hendseth and the author, Thaddäus Hausler. The following text represents the specification as recorded in the NTNU DAIM official archive system.

"e-GOTHAM is a project within the European Artemis framework. The main objective is to implement a new aggregated energy demand model based on the Micro Grid concept. The challenge is to assemble a system that can ensure enough scalability, security, reliability, real time measurements and interoperability.

The student shall develop a local home control, integrated in the existing environment of an e-GOTHAM box. The controller is used to optimise usage of Utilities like Heaters and Boilers with regards to energy cost, energy consumption and comfort. Flexibility and scalability are of main concern for the Controller, as the amount of Utilities, and optimisation techniques used can vary between homes. A proof of concept is to be implemented, working on a test environment simulating an e-GOTHAM box. The simulation shall contain multiple Utilities and optimisation techniques, demonstrating the desired merits." [15]

4.1 Design Considerations

The design goal of the project is specified by the statement "The controller is used to optimise usage of Utilities like Heaters and Boilers with regards to energy cost, energy consumption and comfort. Flexibility and scalability are of main concern for the Controller, as the amount of Utilities, and optimisation techniques used can vary between homes."[15].
The main criteria for the design are as follows:

- Flexibility to schedule including different incentives and Utilities.
- Scalability to include different numbers or Utilities and scheduling algorithms.
- Personalize and tailor the Controller to the individual needs of a local home.
- Abstraction of functionality towards the API.

Due to the flexibility and scalability demands, a basic construct to incorporate these functions has to be designed, capable of easily integrating new features. These features are either new types of Utilities or new techniques of optimising the schedule.
Further, the number of Utilities applied can vary between houses and have to be able to be configured individually.
As the e-GOTHAM infrastructure is still under development after the completion of the controller, the design has to be made as independent from the API as possible.
The goal of the controller is to run in a home of a customer. This means individual decisions have to be taken into account by the design to be able to accommodate customer specific information.

4.2 Proof of Concept

The proof of concept is mentioned by "A proof of concept is to be implemented, working on a test environment simulating an e-GOTHAM box. The simulation shall contain multiple tilities and optimisation techniques, demonstrating the desired merits."[15].
The specific requirements for the proof of concept are described in the following list:

- Different types of Utilities like Heater and Boiler.
- Multiple types of the same Utility possible.
- Multiple goals of optimization.
- Simulate the Controller on a e-GOTHAM box.

To fulfill the specification, a number of decisions have to be made. The Utilities have to be of different types, with different inputs and outputs.
Further the optimisation has to show different techniques of optimisation and be able to operate on the same base to schedule the Utilities in an efficient manner.
The Utilities chosen are Heaters and Boilers, as they are commonly used in houses. Multiple Utilities of the same kind should be possible and different configuration for Utility types is a desired feature.
The e-GOTHAM project specifies a consumption cap, which is fixed in the contract with the home owner. To keep the consumption lower than the cap and the usage of low prices is a feature to be shown. These variables are to be balanced with comfort for the home owner.
For the proof of concept a Raspberry Pi micro controller was chosen to simulate the e-GOTHAM box, containing the desired framework and database but no sensors or other peripherals used in the e-GOTHAM box. The API is provided fully implemented in JAVA and testing data is provided by the e-GOTHAM project.

5 Design and Proposed Solution

Solving the problem described by the specification provides an interesting challenge. The specifications provided are clear in what the focus of the resulting design and proof of concept should lay on.

5.1 From Specification to Design

The controlling aspect of the proposed program can be seen as a type of scheduler, which focuses on multiple aims with different priority, depending on the situation.
One of the first steps to solve this problem is to classify the different problems and fit them into groups. The separation is done in the program by reducing the problem of scheduling multiple utilities to match multiple goals depending on the situation to known problems and solutions.
The first known solution is individual controllers for utilities. Immediate control of powering an Utility, depending on sensor inputs and other information is best done without relying on one shared controller. The resulting Utility controller use previous data of usage and the control is adjusted accordingly.
The second problem is the optimisation of the schedule. The idea is to separate the algorithms for different optimisation techniques and give them just enough information to schedule utilities. The optimisation techniques can then add the individual weights to the decision of the schedule.
To fit these groups together, a top level scheduler manages utilities and optimisation is created. This top level scheduler does not provide smart decisions towards optimising the schedule and neither gives input in the individual control of the utilities.
This separation of parts also calls for slim communication. Three methods of communication into the direction of the utilities and one method for the optimization is the result.
The goals of optimisation in the scheduler are to save energy consumption and cost as a result, while providing comfort to the residents of the local home being controlled. The lines of communication with the utilities are one method to get the estimated consumption for a time, as well as a priority of how important this utility is at the moment for the comfort level of the customer.
These two sources of information and API calls are used to optimise the schedule for a certain amount of time. This is optimisation process is being initiated by the top level scheduler, who then in turn can send the activation and deactivation signal to the utilities.
The top level scheduling has to be activated at a certain time interval, which can be chosen by the customer, but at least every hour. The recommended time between schedules is 10 minutes but all intervals, which result in a natural number of schedules per every hour, is accepted.

5.2 Overview of the Final Design

The final design consists of an architecture, designed to fulfill the needs of the e-GOTHAM project. As part of the smallest embedded units of the e-GOTHAM project, the controller has to be flexible, scalable and smart.
The design presented has very few actors it relies on and thus can be integrated into an existing e-GOTHAM system with ease. As seen in Figure 5.4 the main interaction with the outside is handled by the e-GOTHAM API that collects data and handles the monitoring aspect of the Micro Grid. An overview of the interactions can be found in Table 5.1.

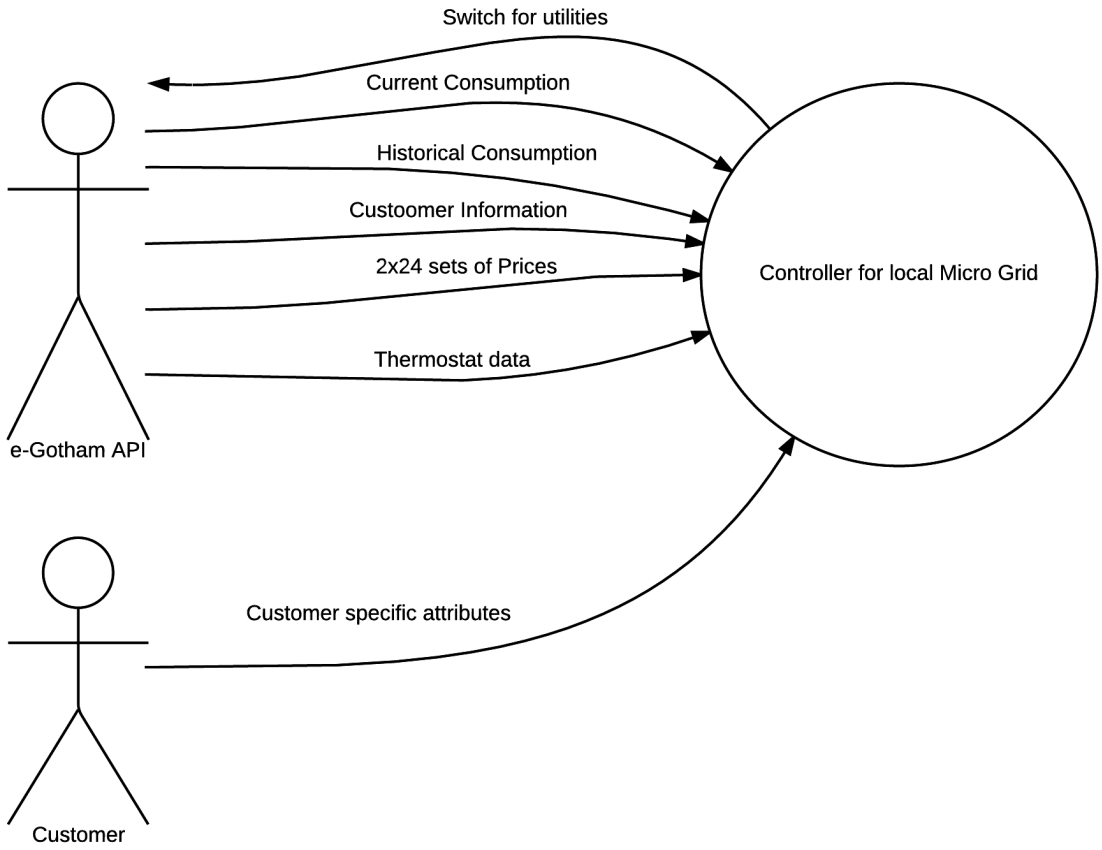

Figure 5.4: Data Flow of the Controller and Actors in the Environment

5.2.1 Actors of the System

The two actors the controller interacts with are the e-GOTHAM API and the customer as shown in figure 5.4.
The information the customer provides the controller with, are of individual nature. The possibility to personalize the behaviour of the controller can be easily extended due to the flexibility of the design. Existing personalisation include the comfort zone temperature, the amount a customer is willing to pay to go over the consumption cap and the scheduling interval.
Further the inclusion of overall personalisation of the system, affecting not only one type of utility, can be achieved. The customer could be able to define zones of behaviour for the system like holiday (e.g. low temperature at all times, boiler off), seasons (Higher temperatures in winter, additions of utilities at these times etc.) or simple preferences in terms of power saving (e.g. how much emphasis between saving energy and staying in the comfort zone).

5.3 Design Entities

In the decision process for the design the concepts of flexibility and scalability were of highest priority. As such the Controller is divided into three parts. These have minimal communication with each other to make implementation of additional modules as easy as possible.
Table 5.2 shows an overview of the three entities.

Table 5.1: Interactions of the Controller with Actors

Actor	Label	Description and Example	Flow Direction
e-GOTHAM API	Switch for utilities	Turn utilities on/off check Status of utility like Boilers or Heaters	Out
	Current consumption	current meter reading	In
	Historical consumption	All recorded consumption of utilities and the full consumption of the system recorded in short intervals	In
	Customer Information	Taken from the contract with the electricity company, e.g. the Consumption Cap	In
	2x24 sets of Prices	Hourly update of pricing data with low/high price	In
	Thermostat data	Temperature, humidity etc. taken from selected rooms	In
Customer	Customer specific attributes	Specific personal data saved locally e.g. what temperature is considered comfortable etc.	In

Table 5.2: Overview of the Design Entities

Name	Function
Scheduler	Creates an optimal schedule
Individual Utilities	Controllers for individual Utilities like heater and boiler
Settings	Provides Information not available through API

5.3.1 Scheduler

The scheduler knows all Utilities as interfaces and treats them equally. Different algorithms are used to activate the needed Utilities according to the specific aims of optimisation.
The scheduler is used to manage all Utilities and optimisation algorithms. The scheduler adds and removes Utilities and algorithms, as well as activates the optimisation process for one iteration and the activation and deactivation of Utilities.

5.3.2 Individual Utilities

The individual Utilities all represent themselves to the scheduler in the same way, even though they can control very different types of appliance. Access to data can be different for each utility.
A utility represents a controller for an appliance like Heaters for rooms or Boilers providing hot water for a household. This can be extended in future work to include other appliances like electric car chargers that promise power savings when properly scheduled.

5.3.3 Settings

The settings contain all information that is not available through the API. As they are stored only locally the settings promise a degree of security and personal preferences, personal data

can be stored here. It is mainly used to store a customer's preferred temperatures and some other price and schedulerelated information.

5.3.4 Data Flow between Entities

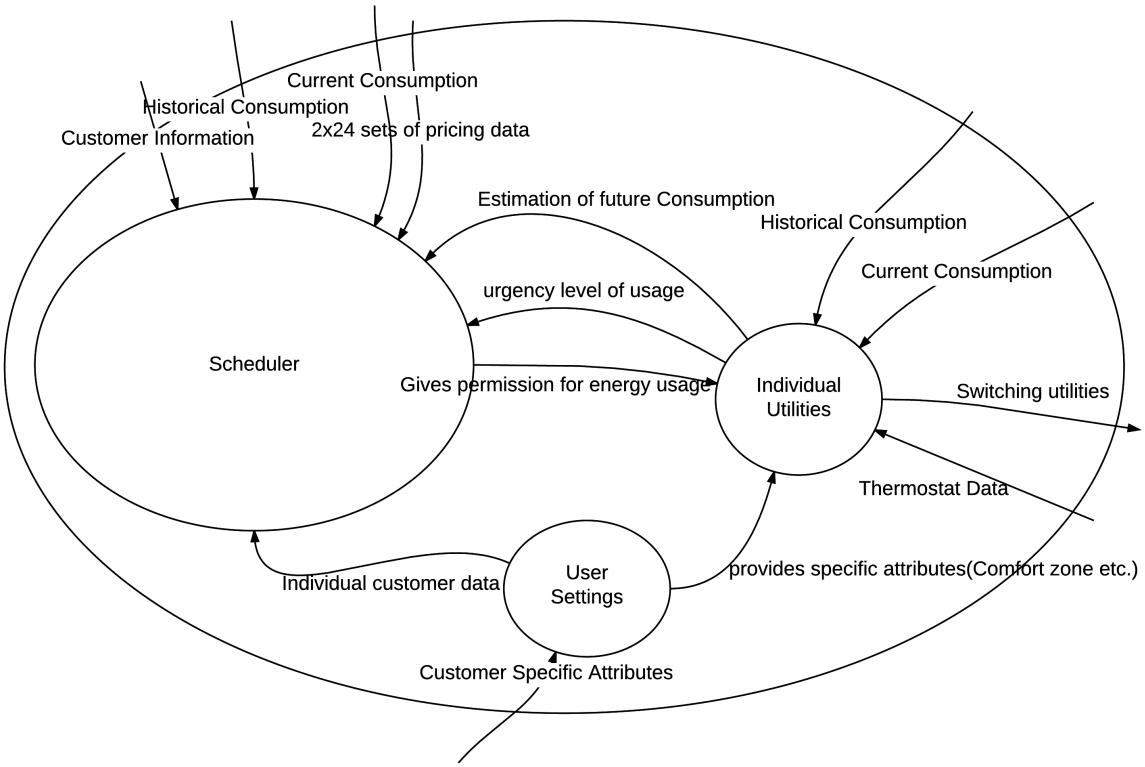

Figure 5.5: Internal Data Flow of the Controller

Data flow between entities and actors on the outside of the controller are chosen to to include only the most essential information. As seen in Figure 5.5, the three design entities are represented as circles and the dataflow as arrows with the direction of where information is flowing to. All arrows breaching the surrounding circle are the same interactions as described in table 5.1.
Within the Controller, the scheduler receives Information from each Utility control about future consumption and an urgency level. The urgency level is used to prioritize Utilities if they reach a critical zone e.g. the temperature drops below the lowest point in the comfort zone or a Boiler has not had the opportunity to heat up water for some time.
The future consumption is used to schedule Utilities. For example if the customer receives a certain energy consumption cap per hour, knowing how much energy is going to be consumed approximately is essential for scheduling utilities in an efficient way.
Individual user settings provide information to the scheduler that personalise the schedule. An example for this information is how much difference between prices a user is willing to accept to use energy beyond a cap. If the difference between the prices is low enough a user might prioritize keeping the temperature within a room at a level more closely to the comfort zone.
The permission for energy usage command is the only outgoing information from the scheduler. This command tells all utilities if they are to be active or inactive for the next period.
The individual Utilities entity receives incoming data from outside of the Controller as seen in

figure 5.5 is described in table 5.1. Additionally the Utilities provide consumption estimation and a priority while receiving commands from the scheduler about scheduled usage of resources. Individual information about the user preferences like comfort zone etc. is provided by the user settings entity.

5.4 Classes and Relationships

A way to show more details about a design is the usage of class diagrams. An overview of the design as described in section 5.3 as class diagram is shown in figure 5.6.

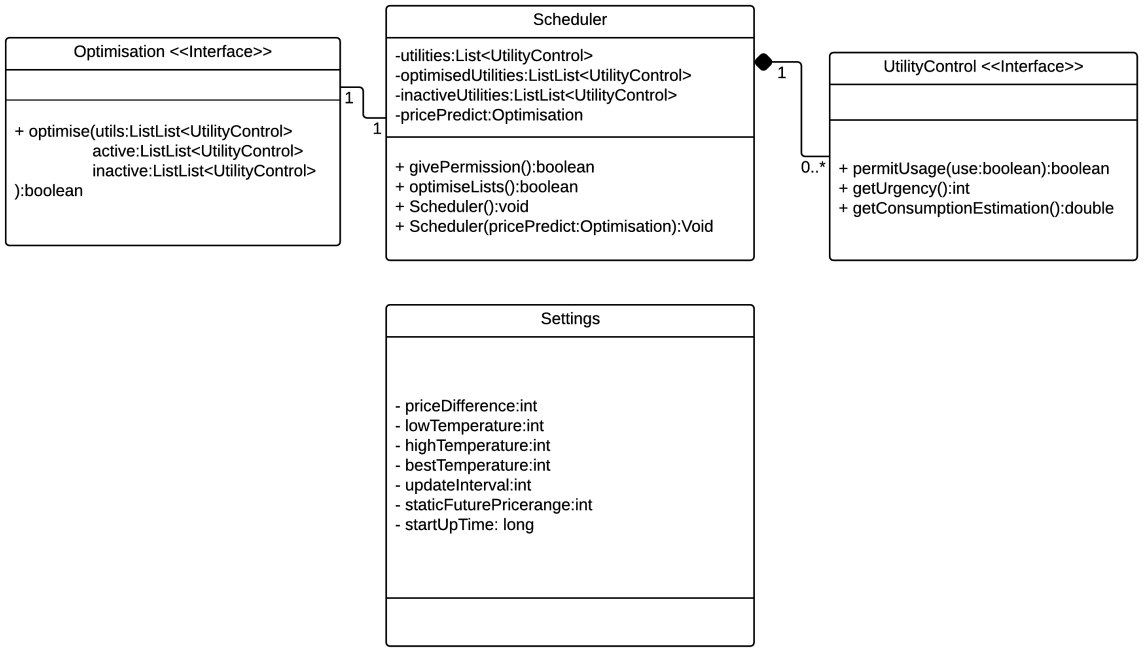

Figure 5.6: Class Diagram with an Overview of the Design

Figure 5.6 shows the entities of the scheduler, the individual Utilities and the settings. The scheduler has one or multiple objects of the utility class, as well as one object of the optimisation class that handles all scheduling activities. The optimisation class operates by creating a list of active and inactive utilities stored in the scheduler class.
Further the scheduler class has methods to start the optimisation and to activate and deactivate the utilities in the appropriate lists.
Within the Utility control interface three methods are defined to cover the information flow. These are controlled by the scheduler class functions and its members.
The settings class has all needed variables for personalisation stored and accessible for all classes and objects that need it.
One of the main focus points of the design is the re-usability and easy modification and inclusion of new parts. As such the design uses Interfaces for the immediate functions of optimisation and control of the different utilities. This adds a level of complexity to the code but enables the relatively easy inclusion of new controls and optimisations.

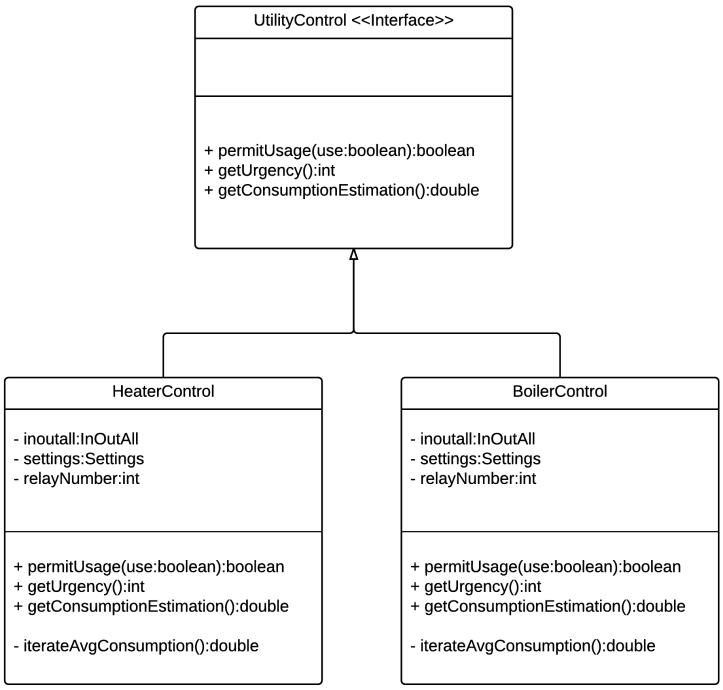

Figure 5.7: Class Diagram showing the Design of the Controller

5.4.1 Controllers

The controllers need to fulfill many different roles depending on the Utility they are controlling. The programming concept used here are interfaces and their implementation. As the example in figure 5.7 shows the interface and the more specialised classes HeaterControl and BoilerControl. The names already give away their function of controlling a Heater or Boiler. Both implement the methods for the interface (permitUsage(), getUrgency() and getConsumptionEstimation()) even though the requirements to create these values differs between them. Each also has a reference to the settings class, the InOutAll class handling all connections to the API and a relayNumber that is used to identify the physical connection in the API.
This implementation also shows a further private function used for creating an average consumption when used.

5.4.2 Optimisation Techniques

Figure 5.8 shows a representation of the design for the optimisation class. As shown the class pricePredictionOptimisation is an implementation of the optimisation interface. It contains the optimise function and a list of techniques for optimisation. This list has at least one sorting technique, represented by the SortingTechnique Interface. The SortingTechnique interface has one function to sort a list of utilities in a certain way.
The addition of another interface with sorting techniques as design choice adds complexity, as an easier way had been to directly access the sorting techniques from the Scheduler. On the other hand it allows the implementation of an optimisation that is not based on sorting the list of utilities but for example a more complex way of controlling a schedule considering many more factors in the equation than the simple implementation done here.

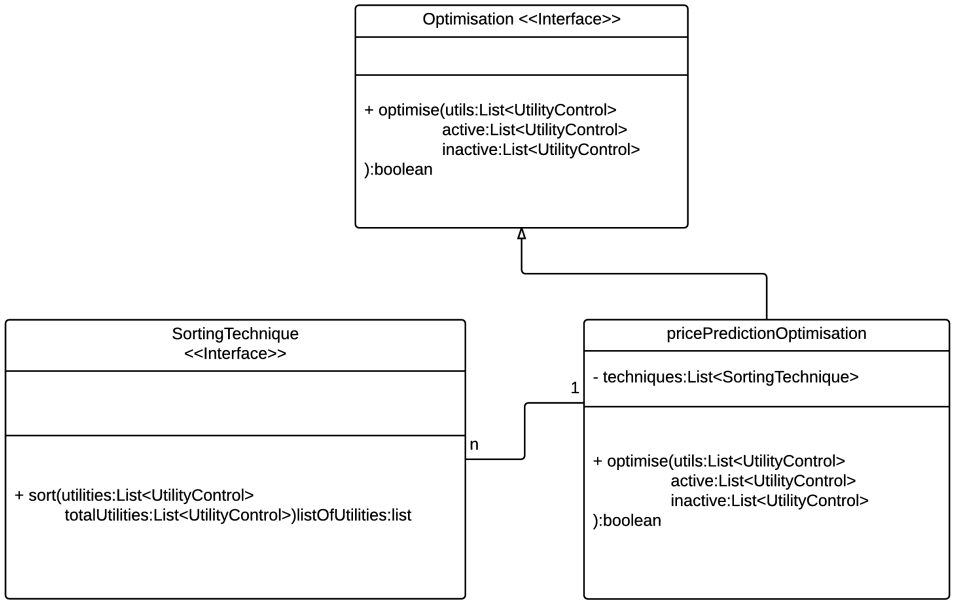

Figure 5.8: Class Diagram Showing the Design of the Optimisation Class

5.4.3 Sorting Techniques

The sorting techniques are implemented as two classes CapSort and PricePredictSort, seen in Figure 5.9. They both have a reference to the InOutAll class for API calls and settings class for individual information not found in the database of the e-GOTHAM box. Further they implement the sort(...) function.
The CapSort class uses the consumption cap, a customer has agreed to, and generates a list out of the available Utilities to not exceed this cap or ignore it, if the difference between prices is acceptable for the customer. It also uses the priorities of Utilities to find a solution.
The PricePredictSort class creates a list of utilities to be active by using the future prices and priorities given by the utilities themselves.

5.5 Settings and Input/Output

These classes are accessible from every other class that needs information. All calls to the API or request for information not generated inside the Controller go through these two classes. The difference between the two is not arbitrary as the Settings handle personalized Information from the client directly that will optimise the Controller for a specific house, while the InOutAll Class handles all calls to the API that will provide information from the database. The database is considered invisible from the view of the controller due to the API handling all calls.

5.5.1 Decoupling from the Design

The design of these two classes comes from an approach to reduce complexity in the program. A call to the API needs “handles“ to different parts of the e-GOTHAM box and in some cases requires certain text strings to receive the correct information.
If a future implementation needs to implement new Controllers, sorting techniques or optimi-

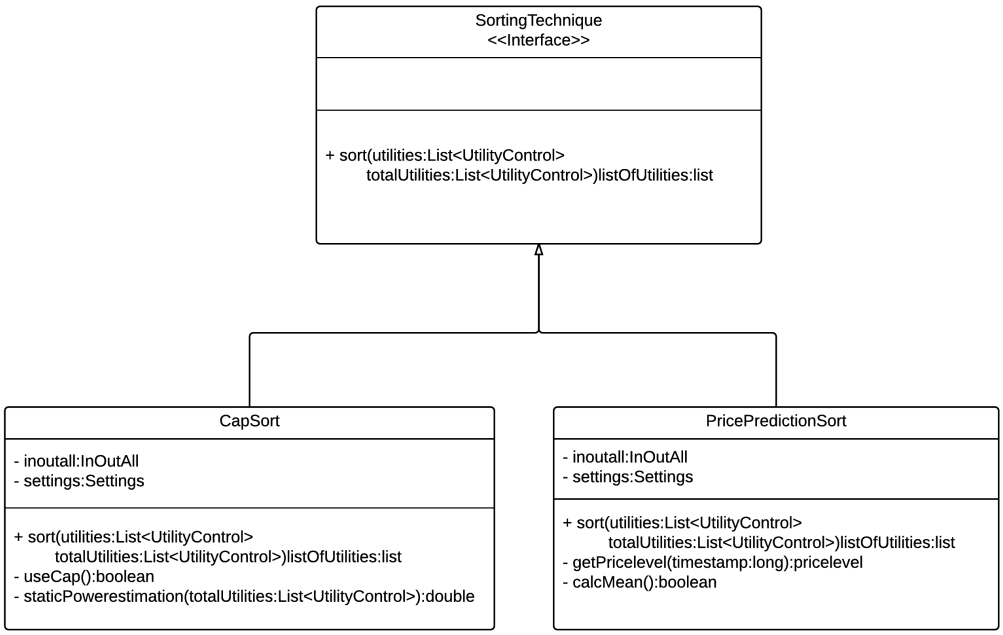

Figure 5.9: Class Diagram Showing the Design of the SortingTechnique Classes

sation techniques they do not need to know everything about the calls made to the API and also find an obvious place to include functions to access new API features. The settings class operates under a similar principle. All information of personal nature to the client is stored in the class and if new features are considered they can be added to the class. Another aspect of the Settings class is privacy. Should the need arise encryption can be added without searching for all relevant calls for information in the design.

5.5.2 Settings

Figure 5.10 shows the settings class on the left side. The class is very simple with getter and setter functions for all variables shown. More detailed explanations for the variables can be found in Table 5.3.

Table 5.3: Variables in the Settings Class

Name	Description
priceDifference	Difference between high and low price the customer is willing to pay for more comfort
lowTemperature	Lowest Temperature for the controls of Heaters
highTemperature	Highest temperature for the controls of Heaters
bestTemperature	Temperature the room should optimally have
staticFuturePricerange	Number of hours in the tariff forecast not to be subject to change
updateInterval	frequency the schedule is to be updated each hour
startUpTime	Time the Program was first started. Is used to find out how much data can be expected to exist in the Database

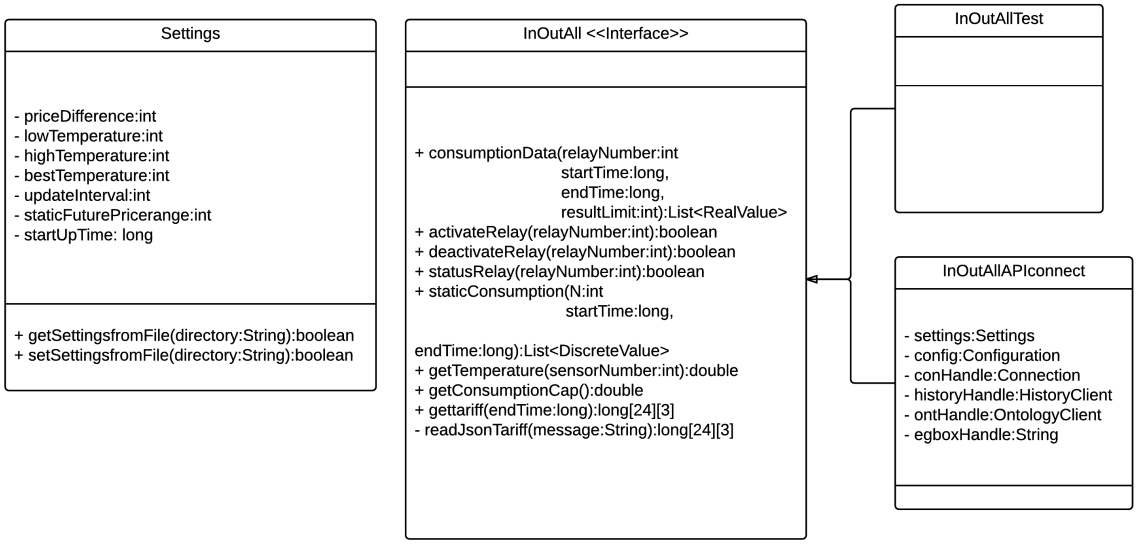

Figure 5.10: Class Diagram showing the Settings and InOutAll classes

5.5.3 Inputs and Outputs

Another class to be found in the controller is the InOutAllAPIConnect class. This class handles all connections to the API. When initialising it generates handles for the different objects needed to access the API. And a reference to the Settings class, as some information from there is needed.
InOutAllAPIConnect is the implementation of InOutAll that regulates and runs the API connection. Table 5.4 lists the functions to be found in InOutAll and gives a short explanation on the right.

Table 5.4: Explanation of the Functions in the InOutAll Class

Name	Purpose
consumptionData	Returns consumption data of different Utilities depending on the relay number from startTime to EndTime with a number of results defined by resultLimit.
activateRelay	Activates a relay.
deactivateRelay	Deactivates Relay .
statusRelay	Returns the state a relay is in (active/inactive).
staticConsumption	Returns a list of data containing the Energy consumption history of the complete system between startTime and endTime with N values.
getTemperature	Returns the current temperature at a sensor.
getConsumptionCap	Returns the hourly consumption cap a customer has signed up to.
getTariff	Returns an array with the newest information about prices in the next 24 hours .

For testing purposes and general abstraction, the InOutAll class is an interface, abstracting the API connection on one hand and including a testing infrastructure on the other. The

InOutAllTest class adds methods to set all returns for the interface. This means, if I need a specific input for testing purposes it can be set directly.

5.6 What are Priorities

Table 5.5: Priority Ranges and Explanation

Prioritiy	Explanation
1	Highest priority,Utility will always be allowed to activate.
2-3	Utility is in need of activation but not critical.
4-5	Utility is running at an optimal point if applicable. Can use activation but does not "need" it.
6-7	Activating Utility will be inefficient but still of value.
8-9	Utility in a state where activation will only trigger possible techniques to store energy while possibly sacrificing comfort
10	Lowest priority, Utility will not be scheduled

Priorities are used as medium of communicating the current status of the utility. In table 5.5 an explanation is shown with priorities on the left and an explanation of the ranges to the right. The descriptions reflect the intent of the priority, as they need to be fine-tuned to work in collaboration with the controllers and the personalisation for the user.
As seen in Table 5.5 the priorities one and ten are handled as absolutes. Priority one will always activate even if the total consumption is already too high. Utilities with other priorities will be postponed in their activation. Priority ten is never scheduled. This priority guarantees controllers the option of turning off completely and no chance of activation.
The priorities four and five represent the optimal state a controller is in. For example a heater has temperatures at the optimal temperature as specified by the user. This is the state the priority price predict class as explained in section 5.8 is urging to achieve, weighting cost and consumption of energy against each other.
The priorities two and three are a gradual step between the highest priority and the optimal point for a utility. It shows the utility's status as not critical but not optimal.
Priorities six to seven relate, that the Utility has no need to activate to achieve the optimal state but can use the activation to achieve a more stable status. For example if the temperature of the room is one degree higher than the optimal temperature for the customer. This is a good state to be in in terms of energy consumption but is not the desired comfort stage.
The two priorities of eight and nine reflect the status of the utility as already past the stage of optimal status and already stable as planned for future inactivity. These priorities are used to activate power storing techniques that have a chance of postponing the next activation but possibly don't bring merit to comfort. An example is a boiler that was active for the last two hours and receives permission to activate. Depending on the model, the size the boiler and the usage it is probable that the water inside is at temperature. Activating the boiler at this point raises the possibility of hot water being provided and can keep it at a more stable temperature.

5.7 Optimsation - PricePredictionOptimisation

Figure 5.11 represents an activity diagram of the optimisation function within the implementation of the Optimisation Interface as described in Section 5.4.2. The Optimisation receives a List with Utilities that have to be optimised. The goal is to return a List of Utilities to be activated, a list of utilities to be deactivated and to not make changes to the full list of utilities. The Optimisation class has one or multiple sorting techniques that can be applied on the list

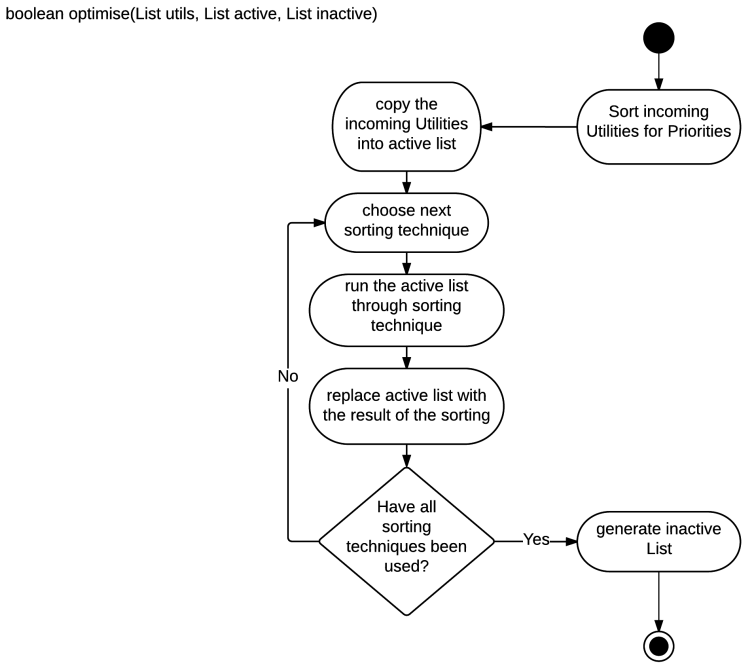

Figure 5.11: Activity Diagram Showing the Implemented Optimisation Technique

of utilities to satisfy different criteria.
In order to make the optimisation a more trivial matter the full List of Utilities is sorted in order of Priorities with the highest priority of one, first. Further, the List of Utilities is copied into a new list, which is subsequently used for the first sorting technique. The order in which the techniques are applied is not of importance in these implementations but as the implementation is focused on re-usability the techniques are used in order of reception (First in First Out).
The following Sorting techniques use the newly optimised list as input and thus refine the active list to match the requirements.
As soon as all sorting techniques have been applied, the function to generates an list of inactive utilities and return the required data.

5.8 Sorting Technique using Price Prediction

The Price prediction sorting technique uses the provided future prices to adjust the active utilities and save money and energy by removing inefficient Utilities from the current iteration.
This technique uses two inputs for its evaluation of the target utilities. On one hand the priority a utility provides is used and on the other hand the price in the active hour and the next hour in relation to the previous prices.

As shown in figure 5.12 the price in the active hour is compared to the past prices. The same comparison is being done with the price in the next hour and used to generate a priority limit for the evaluation period. This process is described in section 5.8.1.
The limit to the priority is applied to the Utilities. Those with lower priorities than the cap are marked incative, those with higher priorities active.

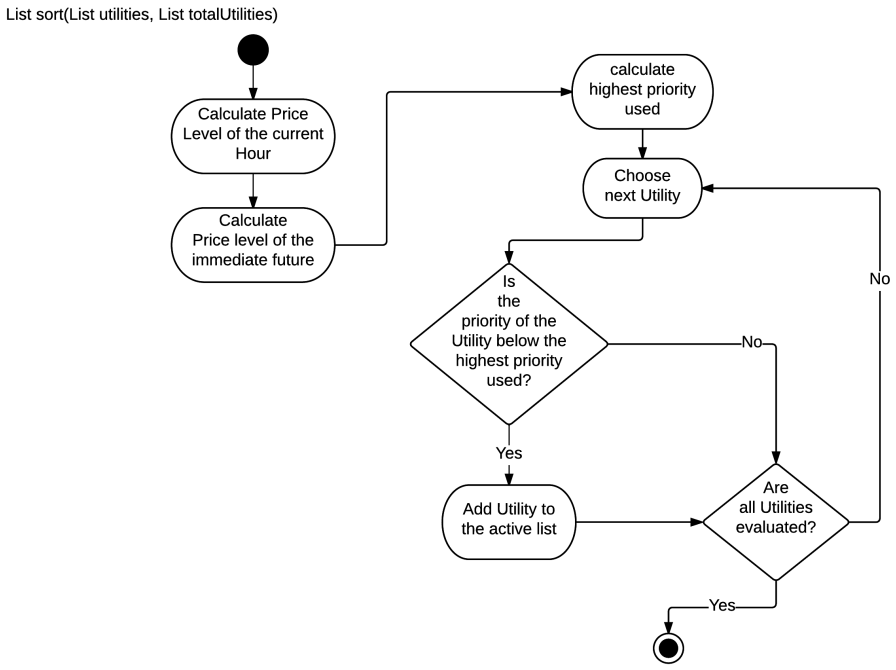

Figure 5.12: Activity Diagram Showing the Implemented Sorting Technique pricePredict

5.8.1 The Relationship Between Prices and Priorities

As the tariff system is still under development, no data was available on the prices. It is assumed that the prices are distributed in a Gauss fashion.
As a sample the prices of the last 100 hours are collected and a mean and standard derivation are calculated. When a comparison between the current or future price is done with this distribution it can be determined as low price (the lowest 17%), average(68% of the values) and a high price (the highest 17%).
With this information the priority limit can be set. Table 5.6 shows the relationship between

Table 5.6: Priority Price Relationship

Pricerange		Now:		
		high	average	low
Future:	high	3	7	9
	average	1	5	5
	low	1	5	5

the values. The x-axis shows the current price level, while the y-axis represents the future price. At the interceptions of the rows and columns a number representing the priority limit can be seen.
Table 5.6 was generated by what would be typically expected but is a candidate for future work as it represents a very good target for personalisation.
In the current design the priority, when both prices are either low or average, has the value of five. The priority of five represents the optimal state of a utility and shows the temperature of

a room is in an optimal setting.
Low prices could be used to save money for example by preheating or topping up a boiler but these possibilities lack a basis and are highly subjective to the user. For example a scenario where the temperature in the room is optimal but the price is low at the moment and the future price is average. Heating the room up now can possibly delay heating in the next hour and save money. Problems with this scenario can be as simple as a user entering the room and opening the window to cool down and thus "wasting" all stored energy.
If it is determined that the current price is high and the future is cheaper (low or average prices) the priority cap is set to 1 which prevents any non-essential utilities from working and thus saving money from not using the high prices.
When the future price is determined as high and the current price as well, priority is set to three, allowing some utilities to be used to keep them closer to the optimal. As an example this allows a heater to be used if the temperature falls towards the lowest point and staying closer to the optimal temperature if the room.
On the other hand if the price in the future is high but the current price is average or low the priority is set to 7 and 9 respectively. This is used to bring all utilities as close to the optimal point and higher in order to prevent usage in the next period.

5.9 Sorting Technique Using the Consumption Cap

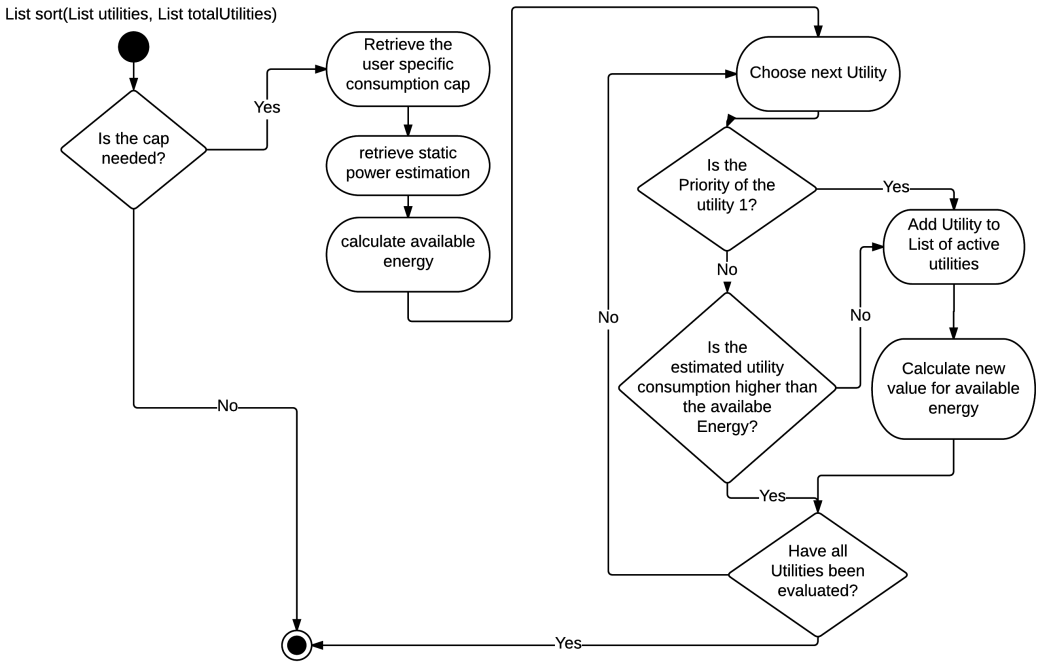

Figure 5.13: Activity Diagram Showing the Sorting Technique CapSort

The consumption Cap sorting technique utilises two pieces of information to save energy consumption, money and provide comfort for the user. The user can specify if a certain difference between the prices above the consumption cap and below is acceptable in order to provide comfort. If the difference between the prices is not acceptable the technique calculates the available energy in a evaluation period and check which Utilities can be active in the period with respect to their priority. This includes the case when a Utility is using too much energy

to stay under cap, but another Utility with lower priority is able to be active without violating the cap. In this case the Utility with lower priority is given active status.
Figure 5.13 shows an activity diagram for the CapSort class. The first evaluation is to check if the consumption cap is to be applied or if the price difference is low enough. If the difference is low enough, no optimisation is needed and the incoming list of Utilities is returned as the result.
If the cap is to be applied the next steps are to retrieve said cap, calculate the estimation for the Utilities that are not controllable from the e-GOTHAM box. This information is used to calculate the available energy for the evaluation period.
As it is assumed that the incoming Utilities are sorted by priority with highest first, the list can be iterated step by step.
Is the priority of the utility of value one they are automatically marked as active and their estimated energy consumption for the period is removed from the available energy.
For the remaining utilities the process is little different. Their estimated consumption is compared to the remaining available energy. Only if it will consume less the utility is marked as active and the estimated amount is subtracted from the available energy.
The sorting is done when all Utilities have been checked if they can be active for the period.

5.10 Utility Controller

The controllers for Heater and Boiler have defined In- and Outputs. The functions they have to fulfill are threefold: Estimate the consumption for the next hour, give a priority on how crucial to the Utilities schedule activation in the next period is and provide a way to set the status to active or inactive.

5.10.1 Toggle Status and Consumption Estimations for Utilities

The functions of activation and deactivation as well as the estimation of the future consumption are the same for both controllers and deeply tied into each other.
An estimation of future consumption can be conducted by analysing past data, sharing similar traits like time of day or temperature. With higher accuracy of the estimation come more calculations, calls to the API and the need for more data. This project is designed to be run on a system with limited system resources. Data the Utilities can provide about usage consumption can only be collected in time and are thus lacking accuracy especially in the beginning of measurement.
The estimation used in the Utility controllers averages the consumption over all while using few API calls and system resources.

Figure 5.14 shows an activity diagram of how the estimation is provided. As the controller is activating the utility, the system time is recorded. As soon as a the controller deactivates the Utility the time is used again to provide boundaries for the API call. A mean is formed out of the acquired consumption data and converted into the desired unit when needed.

5.10.2 Boiler Priority Calculation

The assumption for a boiler is that all data provided concerning this utility is the consumption. Most modern boilers have a controller installed and can heat and top off their load using sensors inside the tank.
A further assumption made is that a functional boiler can be considered in an optimal state, if it has been active for a certain amount of time. This is reflected in the function calculating

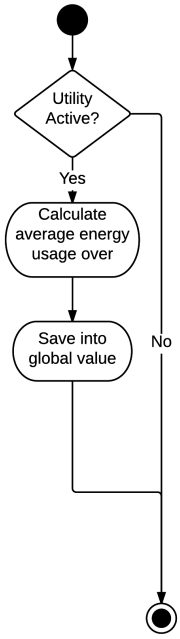

Figure 5.14: Activity Diagram Showing an Estimation of Future Utility Consumption

the priority. The controller monitors the previous two hours. If the boiler has been active for more than half the time the priority returned is ten. A usage of 25% to 50% is considered optimal and a priority of five is returned. 25% to 10% is marked as a priority of 3 and below 10% activity is returning a one.

5.10.3 Heater Priority Calculation

The priorities of the Heater are being generated by applying priorities to temperature ranges. The customer specifies the three temperature fix points. The borders of the temperature range are specified by the minimal and maximum temperatures and also names an optimal temperature. Figure 5.15 shows the temperature range and the resulting priorities. The black arrow shows a temperature scale while the minimum, optimal and maximum temperatures are marked by arrows pointing up. White arrows represent the range in which the underlying priority number is used. The figure shows that below the minimum temperature the priority will be one. Between the lowest temperature and the optimal temperature, enlarged by a small temperature range, a priority of two is applied. The optimal range has a priority of five, between optimum and highest temperature a priority of six is applied and finally all temperatures higher than the maximum show a priority of 10.

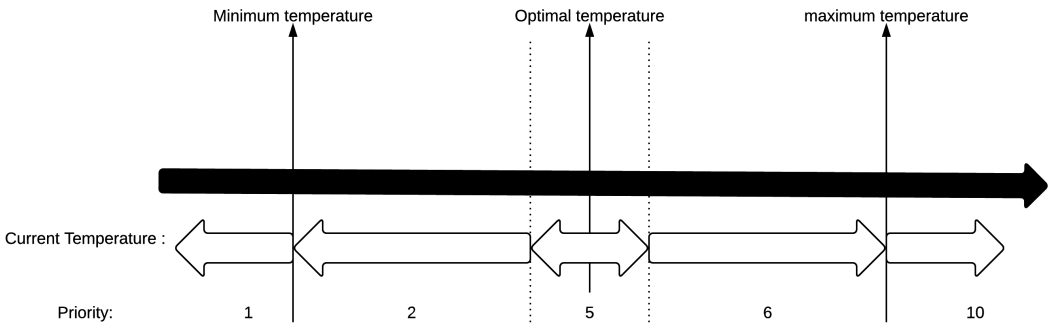

Figure 5.15: The Heater Temperature Range Correlating with Priorities

6 Implementation

The implementation of the design and proof of concept shows an approach to create a structure needed to fulfill the specifications. The following chapters show the different classes and methods with their functionality and, in some cases, the actual Java code.
The full code of the proof of concept and the design implementation can be found in the files appended to this report.

6.1 Programming Language and Tools

The programming language used in the Implementation, is Java. The choice of language is not done by the Author but is a requirement when working with the e-GOTHAM API, as it is the best supported language.
For the practical part of the implementation, the Integrated Development Environment (IDE) used is the Netbeans environment provided by Oracle, which is open source available. The IDE included support for a GIT repository for version control, which is hosted on the e-GOTHAM server at NTNU.
The Setup the project is written in includes a Windows 7 PC, running a virtual machine (Virtual Box) with Ubuntu Linux and the mentioned Netbeans. The reason for the virtual machine is the protobuf Java library which is unfortunately not available for windows. The hardware is provided by NTNU and SINTEF.
The tests and proof of concept is tested on a Raspberry Pi connected to the University network via the inbuilt network port and file transfer is handled via the SPC program on Linux and WinScp on Windows. The direct connection to the Raspberry Pi is maintained through the PuTTy program in its Windows version.
Further java packages used are the commons maths package and the json simple package. The Maths package is used for calculation of the mean and standard deviation in the price prediction, while the json simple package plays an integral role at decoding information provided by the API All relevant software tools and versions can be found in table 6.7

Table 6.7: Used Programs, Packages and their Versions

Program	Version
e-GOTHAM:	
egotham -common, -sdi, -communication	1.1
Other packages :	
Commons maths	Commons-math3-3.5.jar
json.simple	org.json.simple-0.4
IDE:	
Netbeans	Netbeans 8.0.2 Build 201411181905
protobuf plugin	protobuf-java-2.6.1
Java	jdk 1.7.0_79; OpenJDK 64-Bit Server VM 24.79-b02
Operating systems:	
Ubuntu	10.04 LTS, Kernel: 3.16.0-30.generic
Raspberry Pi	Debian 3.12.9-1+rpi3
Windows	Win 7 Enterprise Service pack 1 64 bit
PuTTy	0.64
WinSCP	5.7.2 Build 5316
Virtual Box	4.3.26 r98988

6.2 Implementation - Scheduler

As described in section 5.3 the Schedulers purpose is to hold all objects needed for the scheduling and give permissions to utilities if they are to be active.

6.2.1 Variables and Constructor

For this purpose the scheduler variables as seen in 6.1 are private Lists for the full list of utilities, a list for the active number of utilities and the inactive utilities as well as an object for the optimisation. All variables are private and have appropriate setter and getter methods. The setter methods for the lists ensure they are empty before new objects are added and the lists are initialised in the constructor to ensure correct handling.

```
1 public class Scheduler {
2
3     private List<UtilityControl> utilities;
4     private List<UtilityControl> optimisedUtilities;
5     private List<UtilityControl> inactiveUtilities;
6
7     private Optimisation pricePredict;
```

Listing 6.1: Variables in the Scheduler Class

6.2.2 Set Permissions for Utilities

To set Utilities to active and deactivate the non Scheduled set is a public method of the Scheduler.
As a primary precaution the method first checks if the active and inactive utility lists add up to the full number of utilities, as an incomplete list would result in problematic scenarios and can indicate that the lists are not up to date. This is done by simply comparing the amount of objects in the active and inactive list with the number of objects in the full list of utilities.
As return value a Boolean value signals the successful activation and deactivation of all utilities.

6.2.3 Creation of the Schedule

The method optimiseLists() calls the optimisation object with the appropriate variables. The return value is a Boolean value that signals if the process was successful.

6.3 Implementation - Controller

The Controller is known as the class UtilityControl in the implementation. The class is an interface with the three methods boolean permitUsage(Boolean) to change the status of a utility, a method getUrgency() with return value Integer to get a priority and finally a getConsumptionEstimation() method that returns the future consumption of a Utility as a double value in watt*hour.
In order to sort the Utilities in an efficient way a class UtilityComparator is implemented to override the compare method. Listing 6.2 shows the implementation. Two Utilities, a and b are being compared. A positive value is returned if a¿b, negative if a¡b and 0 if they are equal. This seems counter intuitive, as a priority of 1 is considered “higher“ than one of 5 but is of benefit when using the oracle implementation of a sorting algorithm.

```
public class UtilityComparator implements Comparator<UtilityControl> {

    public UtilityComparator() {
    }
    @Override
    public int compare(UtilityControl a, UtilityControl b ) {
        return a.getUrgency()-b.getUrgency();

    }
}
```

Listing 6.2: Implementation of Comparator

6.4 Implementation - permitUsage and getConsumptionEstimation

The two methods are the same for both implementations, in HeaterControl as well as BoilerControl.
The method permitUsage has as main purpose the activation or deactivation of a utility which is done by calling a function in the API. Listing 6.3 shows an implementation of the method. As shown the Boolean variable "use" represent the state the utility is supposed to be in.
If the utility status is to be active or inactive, the appropriate API function is called. The generated returnvalue reflects if the status of the utility actually changed or if it has had the same status as before.
On activation the method checks, if there is a measurement going on. If this is not the case it sets the ongoingMeasurement value to TRUE and records the current timestamp in the start variable.
On deactivation, and if there has been a measurement ongoing, the method resets the ongoingMeasurement value to FALSE and records the current time in the stop variable.

```
    @Override
    public boolean permitUsage(boolean use) {
        boolean returnvalue;
        if (use) {
            try {
               returnvalue =this.inoutall.activateRelay(relayNumber)) {
                if (!ongoingMeasurement) {
                    ongoingMeasurement = true;
                    start = Clock.getInstance().currentTimestamp();

                }
            } catch (Exception ex) {
                Logger.getLogger(HeaterControl.class.getName()).log(Level.SEVERE
    , null, ex);
            }
        } else {
            try {
                returnvalue = this.inoutall.deactivateRelay(relayNumber))
                if (ongoingMeasurement) {
                    ongoingMeasurement = false;
                    stop = Clock.getInstance().currentTimestamp();
                    avgConsumption = iterateAvgConsumption();
                    returnvalue=true;
                }
            } catch (Exception ex) {
                Logger.getLogger(HeaterControl.class.getName()).log(Level.SEVERE
    , null, ex);
            }
```

```
27          }
28          return returnvalue;
29      }
```

Listing 6.3: Implementation of permitUsage Method

The method iterateAvgConsumption is called after the times have been recorded and accumulate all consumption data from the relay. A mean value is formed out of the data in addition to the old mean value and recorded in the avgConsumption variable.
This value is used in the getconsumptionEstimation to calculate the current average consumption per hour and return as double value.

6.5 Implementation - Heater Controller

The HeaterControl class is an implementation of the UtilityControl Interface. As such it implements the three previously mentioned methods permitUsage(Boolean), getUrgency() and getConsumptionestimation.

6.5.1 Variables in HeaterControl

The variables needed in the HeaterControl class are named and specified in table 6.8. RelayNumber, sensornumber and the settings object have appropriate setter and getter methods, all others are initialised in the constructor or elsewhere.

Table 6.8: Variables in Heater Control Class

Type	Variable	Purpose
Settings	settings	Access to consumer specific information
InOutAll	inoutall	Access to API functions
int	relaynumber	Individual relay number relating to the socket it is plugged into
int	sensornumber	number of the thermostat the relating temperature measurements are to be taken from
long	start	e-GOTHAM timestamp marking the start of a measurement
long	stop	e-GOTHAM timestamp marking the end of a measurement
double	avgConsumption	value for the average consumption in watt*second
boolean	ongoingMeasurement	marks if the heater is currently active

6.5.2 getUrgency Implementation

The getUrgency method as seen in listing 6.4 uses a very simple control algorithm. First all external values needed are collected. These include the high, best and lowest temperature out of the settings class and the current temperature measurement from the appropriate sensor. Further the Priority is calculated by checking where on the temperature spectrum the current temperature is to be found.

```
1 public int getUrgency() {
2
3         int bt = this.settings.getBestTemperature();
4         int mint = this.settings.getLowTemperature();
5         int maxt = this.settings.getHighTmeperature();
```

```
6
7           double ct =this.inoutall.getTemperature(sensornumber);
8
9           if (ct < mint) {
10              return 1;
11          } else if (mint < ct && ct < (bt - 1)) {
12              return 2;
13          } else if (bt - 1 < ct && ct < bt + 1) {
14              return 5;
15          } else if (bt + 1 < ct && ct < maxt) {
16              return 6;
17          } else if (maxt < ct) {
18              return 10;
19          } else {
20              return 3;
21          }
22
23      }
```

Listing 6.4: Implementation getUrgency in HeaterControl

6.6 Implementation - Boiler Controller

BoilerControl is an implementation of the UtilityControl Interface and is used to regulate the usage of a water boiler.

6.6.1 Variables in BoilerControl

The variables used in the BoilerControl class are the same as in HeaterControl seen in table 6.8 with an addition of the activityTimestamps array of long values. The size of the array is determined by the length of the activity we are looking at, in this case two hours, in minutes divided by the update interval, given by the settings class in minutes.
It is initialised in the constructor and filled with zeros. The variable is needed in the priority prediction method of the BoilerControl class.
For the same method a double value percentageOfUsage is used. It reflects the percentage of how often the Boiler was in use in the length we are looking at in the activityTimestamps.

6.6.2 getUrgency Implementation

The urgency value is calculated by using the percentageOfUsage variable, which saves the percentage the Boiler has been active in the last two hours. According to a table it returns ten for more than 50% usage, between 50% and 25% a five is returned while 25% to 10% returns a three. Everything below 10% returns a one and requests immediate activation.
The percentageOfUsage is calculated by calling a private method calculatePercentage(). It is called every time the Boiler is activated or deactivated. The implementation can be seen in Listing 6.5.
The implementation relies on the activityTimestamps array. The array is shifted to the left and if the Utility is active at this point a timestamp is added. If not, the value zero is written into the last place of the array.
The percentage of usage is calculated by counting the number of zeros in the array, divide it by the length of the array and subtract the result from one.

```
1 private void calculatePercentage() {
2
3           long currentTime = Clock.getInstance().currentTimestamp();
```

```
4         int count = 0;
5         this.percentageOfUsage = 0;
6         for (int i = 0; i < this.activityTimestamps.length - 1; i++) {
7             this.activityTimestamps[0 + i] = this.activityTimestamps[1 + i];
8
9         }//shifting the array once to the left
10        try {
11            if (this.inoutall.statusRelay(relayNumber)) {
12                this.activityTimestamps[this.activityTimestamps.length] =
    currentTime;
13            } else {
14                this.activityTimestamps[this.activityTimestamps.length] = 0;
15            }// adds a timestamp at the last space of the array, 0 if the relay
    is off
16        } catch (Exception ex) {
17            Logger.getLogger(BoilerControl.class.getName()).log(Level.SEVERE,
    null, ex);
18        }
19
20        for (int i = 0; i < this.activityTimestamps.length; i++) {
21            if (this.activityTimestamps[i] == 0) {
22                count++;
23            }//counts the amount of times the relay was off
24
25        }
26        this.percentageOfUsage = 1 - (count / this.activityTimestamps.length);
27
28    }
```

Listing 6.5: Implementation calculatePercentage in BoilerControl

6.7 Implementation - Optimisation

The optimisation class is an interface providing the optimise(...) method. The method receives three Lists with UtilityControl objects. The utils list contains all utilities in the system, all active utilities are in the active list while all inactive utilities are stored in the inactive list.
The implementation of the Optimisation interface is called PricePredictOptimisation.
The method of creating a schedule is by applying different sorting algorithms on the list of utilities. The sorting algorithms are explained in section 6.8.
To apply all sorting algorithms the list of utilities is first sorted to order the utilities from the highest priority to the smallest.
The next steps involve stepping through the list of SortingTechnique objects and calling their sort(...) method. For the first technique the list of utilities is the full list of utilities. All following techniques get the result of the previous sorting as inputs, as well as a full list of all utilities involved.
The resulting list is sorted and written into the active variable, while the inactive variable is generated from the full list minus the active list.

6.8 Implementation - SortingTechnique

The SortingTechnique class is an interface as well and provides the sort(...) method. The method has a Boolean value as output, which indicates if the execution was done correctly. As inputs, the method receives two lists of UtilityControl. The first list contains the number of utilities to be sorted and the second list reflects the full list of registered utilities.

6.9 Implementation - Price Prediction

PricePredictSort is a class implementing the SortingTechnique interface and as such implements the sort(...) method.
The method uses the prices provided by the API and the priorities provided by the utilities to determine which utilities should be active.

6.10 Variables in PricePredictSort

Table 6.9: Variables in PricePredictSort

Type	Variable	Purpose
Settings	settings	Access to consumer specific information
InOutAll	inoutall	Access to API functions
double	mean	mean value of past prices
double	std	standard deviation of past prices

Table 6.9 lists the variables used in the PricePredictSort class. The first two inoutall and settings are used to access information from outside the Controller environment like API calls. mean and std are the mean and standard derivation of the past prices, used to determine if a price is high, average or low relative to the past prices.

6.10.1 Evaluating the Prices

In order to use current and future prices as guidelines for scheduling a measure has to be performed of how the prices relate to past prices. The method uses only the lower prices out of the provided tariff.

```
1  private enum priceLevel {low, high, average}
2
3  private priceLevel getPricelevel(long timestamp) throws Exception{
4      long[][] tariff = new long[24][3];
5      tariff = inoutall.gettariff(timestamp);// price at timestamp
6      calcMean();
7      if (tariff[0][1] <(this.mean-this.std))
8      {return priceLevel.low;}
9      else if(tariff[0][1] >(this.mean+this.std))
10     {return priceLevel.high;}
11     else
12     {return priceLevel.average;}
13 }
```

Listing 6.6: Implementation getPriceLevel and priceLevel Enum

In order to perform this, it is assumed the prices can be found on a Normal Distribution. A further assumption is made that 100 values are enough to get a reasonable distribution.
In the implementation a enum is used to reflect the relative level of a price. Listing 6.6 shows the implementation. The values can be low, average or high.

The listing also shows the private method to determine a price level at a certain timestamp, called getPricelevel(...). The method takes an e-GOTHAM time stamp and returns the price level at this point.

In order to perform this action the tariff at the time stamp is retrieved from the API and the Mean and Standard Deviation is calculated. The latter is done by using the math3 library available in the apache commons API. In order to calculate the mean a DescriptiveStatistics object is created and the previous 100 price values are used for the calculation. In order to limit API calls, the FuturePriceRange in the Settings class is used to read the prices out of the tariff files.
The price in question is then compared to the mean and standard deviation to find out if it is high, average or low.

6.10.2 Priorities vs Price level

Priorities of the utilities are used to determine whether the utility is to be active. To achieve the limit of which priorities should be active a matrix is used as described in table 5.6. The method used to do these steps is called sort(...). The method first establishes what pricelevel the next hour and the current hour have and establishes the mentioned priority maximum. The utility priorities are then compared to the maximum and written into the return value.

6.11 Implementation - Consumption Cap

The consumption cap sorting technique, called CapSort, utilises two pieces of information to save money and provide comfort for the customer. On one hand each customer agrees on a consumption cap for each hour. When more Energy is used the price goes up.
On the other hand the customer can specify what price difference between high and low price is acceptable as a tradeoff for more comfort and less efficiency in most cases.

6.12 Validation of the Usage of the Consumption Cap

The private method useCap() evaluates the need of the consumption cap and returns a Boolean TRUE if the cap is to be used.
In order to find this, the method calls the API to obtain the current prices. If the difference between the prices is smaller or the same than the limit a customer has set, a FALSE is returned. Listing 6.7 shows the implementation with the variable tariff receiving the prices and the PriceDifference variable providing the difference acceptable for a customer.

```
1     private boolean useCap() {
2         try {
3             long[][] tariff = inoutall.gettariff(Clock.getInstance().
    currentTimestamp());
4
5             if ((tariff[0][2] - tariff[0][1]) <= settings.getPriceDifference())
    {
6                 return false;
7             }
8
9         } catch (Exception ex) {
10            Logger.getLogger(CapSort.class.getName()).log(Level.SEVERE, null, ex
    );
11        }
12        return true;
13    }
14 }
```

Listing 6.7: Retrieving the Validity of the Consumption Cap

6.12.1 Static Energy Estimation

The CapSort class needs a way to get an estimation of how much energy was consumed in the last hour, that did not involve a controllable utilitiy. The API does not provide a direct way to find the static energy consumed in the last hour.

6.12.2 Sorting using the Consumption Cap

In order to use the cap, the available energy for activating utilities is needed. For this purpose a iteration counter has to be established. Every time the sort(...) method is called, a iteration counter is incremented until it reaches the maximum amount of iterations for this hour, as defined in settings. If the iterCounter variable is zero, a starting energy is saved in startEnergy. This value is used later to determine how much energy was consumed so far this hour.
Listing 6.8 shows the relevant part of the implementation of the sort() method. The first step is used to evaluate if a cap is needed as described in section 6.12. Further the iteration counter is evaluated and iterator reset. A reset also marks the start of a new evaluation period and the startEnergy variable is reset.
The calculation of of the available energy for this interval. This is done by subtracting the energy consumed since the start of the hour from the cap. This represents the total energy available for the rest of the hour. This is divided by the rest of the iterations for the hour and finally the static consumption for the iteration period is subtracted.
The resulting available energy is used to determine which Utilities are to be active, by testing every utility, highest priority to lowest, if they consume less than the available energy. Priority one Utilities are prioritized and can push the past the cap and cause high consumption. Every Utility that does not match these requirements is removed from the list of utilities. The return value is the list of utilities to be active.

```
1     if (useCap()) {
2
3                     if (iterCounter >= iterations) {
4                         iterCounter = 0;
5                         startEnergy = currentEnergy;
6                     } else {
7                         iterCounter++;
8                     }
9
10
11                  // calculate available power for interval
12                  availableEnergy = cap - (currentEnergy - startEnergy); // the
    rest of the available energy for 1h
13                  availableEnergy = availableEnergy / (iterations - iterCounter);
    //divided by the amount of iterations left this hour
14                  availableEnergy = availableEnergy - staticEnergy / iterations;
    // subtract the available static energy available for one iteration
15
16                  // find which utilities can be active at this point
17                  for (UtilityControl util : utilities) {
18                      tmpEnergy = util.getConsumptionEstimation(); // consumption
    estimation for the utility in W*h
19                      if (!(util.getUrgency() == 1) && tmpEnergy > availableEnergy
    ) {// if the utility doesn't have prio 1 or consumes more than the cap remove
     it
20                          returnValue.remove(util);
21                      } else {
22                          availableEnergy -= tmpEnergy/iterations;    //available
    energy is reduced by the running utility
23                      }
```

```
                }


            }
```

Listing 6.8: Sorting Technique Utilising the Consumption Cap

6.13 API connection - InOutAll

The InOutAll class handles all connections to the API and abstracts calls for the as methods with simpler interfaces.

6.13.1 Variables in InOutAll

Table 6.10 shows the variables used in the InOutAll class. They specify connections to the different parts of the e-GOTHAM box, which contains a database with consumption data etc. and sensors in connection to the utilities.

Table 6.10: Variables in InOutAll

Type	Variable	Purpose
Settings	settings	Access to consumer specific information
Configuration	config	default configuration for the API connection
Connection	conHandle	Handle for the connection to the API
HistoryClient	historyHandle	Handle for the History Client
OntologyClient	ontHandle	Handle for the Ontology Client
String	egboxHandle	String with the name of the e-GOTHAM box the software is running on

6.13.2 Activation and Deactivation of a Relay

Listing 6.9 shows the code for the activation of a relay. Unnecessary activation and deactivation commands are not encouraged, as they can produce unknown effects. The method tests for the proper status of the relay and returns a FALSE value if the relay is already in the desired state. If the relay still needs to be activated or deactivated a Command object is found by calling the ontHandle.find(...) method with the right parameters. Further a Source object is needed to send an activation command via DiscreteValue(1).
The deactivation method is very similar, except it checks the relay for the opposite case (is it already inactive) and sends a DiscreteValue(0) to the Source object.

```
1   public boolean activateRelay(int relayNumber) throws Exception{
2
3
4       if (!statusRelay(relayNumber)){
5           Command relay = this.ontHandle.find(Command.class, this.egboxHandle
    + ".relay." + String.valueOf(relayNumber) + ".command");
6           Source relaySource = new Source(conHandle, relay);
7           relaySource.send(new DiscreteValue(1));
8           return true;
9       }
10      return false;
11  }
12
```

Listing 6.9: Activation of a Relay

6.13.3 Status Request on a Relay

Listing 6.10 shows the method statusRelay(...). The public method returns the current status of a relay as a Boolean value.
In order to achieve this, the method creates a Discrete object with the help of the ontHandle The object is used to create a SeriesQuery object which, with the current time as end time and one as resultlimit, will return the current status of the relay.

```
1        public boolean statusRelay(int relayNumber) throws Exception{
2
3        Discrete relay = this.ontHandle.find(Discrete.class, this.egboxHandle+ "
    .relay." + String.valueOf(relayNumber) + ".state");
4        SeriesQuery query = this.historyHandle.createSeriesQuery(relay);
5        query.setEndTime(Clock.getInstance().currentTimestamp());
6        query.setResultLimit(1);
7        List<DiscreteValue> execute = query.execute();
8        return (execute.get(0).v == 1);
9    }
10
11
```

Listing 6.10: Status of a Relay

6.13.4 Consumption Data of a Relay

Utilities are connected to relays. These relays make the measurement of power and the activation/deactivation possible.
The consumptionData(...) method requires a relay number, a start time, a end time and a result limit. This is the general method of obtaining data from the API.
Listing 6.11 shows how the retrieval of consumption data out of the API is handled.
As the database is filled with data over time, there might not be any initial data available. This is compensated by checking the requested period and resetting start time and end to appropriate values. The most extreme case is when both times are lower than the startup time recorded in the Settings class.
The data request itself starts off with the creation of an Analog object, created with the proper string representing the needed data, including the relay number, the handle for the box itself and more identifiers.
A SeriesQuery object is created with the Analog object as input. The query handles the required methods to access the database and, in this case, returns a List¡RealValue¿. The query has different variables to set the start- and end-times, as well as a result limit.
The result limit specifies how many results are written into the return value. A zero will return all recorded values for the specified period, a negative result limit of N will return the N first recorded values, a positive N returns the N latest results.

```
1  public List<RealValue> consumptionData(int relayNumber,long startTime, long
     endTime, int resultLimit){
2        List<RealValue> result = new ArrayList<>();
3        if (startTime < settings.getStartUpTime())//not enough data is existing
4        {
5            if (endTime < settings.getStartUpTime()) {//interval is entirely
     before startTime.
6                startTime = 0;
7                endTime = settings.getStartUpTime();
8            } else {
9                startTime = settings.getStartUpTime();
10           }
11       }
```

```
12          try {
13              Analog consumption = ontHandle.find(Analog.class, this.egboxHandle +
    ".relay."+String.valueOf(relayNumber)+".power");
14
15              SeriesQuery query = historyHandle.createSeriesQuery(consumption);
16
17              query.setStartTime(startTime);
18              query.setEndTime(endTime);
19              query.setResultLimit(resultLimit);
20
21              result = query.execute();
22          } catch (Exception ex) {
23              Logger.getLogger(InOutAll.class.getName()).log(Level.SEVERE, null,
    ex);
24          }
25          return result;
26      }
```

Listing 6.11: Retrieving Consumption Data from a Relay

6.13.5 Overall Consumption

The overall consumption is the sum of all energy consumed and registered by the electrical meter. This includes the dynamic and static consumption. The code to obtain the data is much the same as described in section 6.13.4 with the exception of the Analog object, which is replaced by an Accumulator object, and the string needed to create the Accumulator object.

6.13.6 Read Temperature Data

The Temperature data is obtained by using the same steps as described in section 6.13.4. The only changes to the code can be found with the handling of insufficient data, as the temperature sensor data is not accessed with a time stamp and thus, as long as a thermostat is attached, there will be data. Another difference is the string used to create the Analog object.

6.13.7 Reading the Tariff

The tariff is a construct of 2x24 prices, used in PricePredictSort to schedule utilities.
The API saves the tariff as a JSON string in the Database. The JSON format is described in 3.5.
A StringMeasurement is the type used to query the tariff from the API. As seen in previous implementation the ontology handle is used to find the correct StringMeasurement, with the help of the correct string ".tariff". Further steps include setting a query for with the correct time stamp as end time and as we only need one measurement the result limit is applied.
The resulting string is read by a JSON parser and converted into a long array with one column having timestamps for the prices, the second column is for the low prices, the third for the high prices. This result is returned to the caller.

```
1
2       StringMeasurement jsonTariff = ontHandle.find(StringMeasurement.class,
    egboxHandle + ".tariff");
3       SeriesQuery query = historyHandle.createSeriesQuery(jsonTariff);
4       query.setEndTime(endTime);
5       query.setResultLimit(N);    // gets the latest tariff
6       List<StringValue> result = query.execute();
7       String s = result.get(0).v;
8
```

```
9         returnValue = readJsonTariff(s);
```

Listing 6.12: Implementation of the Tariff Retrieval from the API

6.13.8 Decoding a JSON String

JSON Strings are used to encode information from the database in text form. The medium by which the data is encoded is as a string with fixed rules as to how to decode it. The example shown in listing 6.13 is the retrieval of the tariff data from the database.
In the example a JSON string is read from the database and reafJsonTariff(...) is called with the string as input. An example for the string can be found in the appendix under section B.16. A parser object from the json-simple library is created. A JSON object is created as well, and the message is given to the object as input, after parsing it.
In this particular case, the decoding has to begin by finding the first hour identifier and its value. The first hour is an e-GOTHAM timestamp with the first hour the string provides a price for.
Subsequently the string in the JSON object can now be searched for the hour and the prices attached to it. This is repeated until all values are retrieved and saved in the long array functioning as return value, which is returned to the caller in the end.

```
1
2
3     private long[][] readJsonTariff(String message) throws ParseException {
4         long[][] returnValue = new long[24][3];
5
6         JSONParser parser = new JSONParser();
7         JSONObject obj = (JSONObject) parser.parse(message);
8         JSONObject tmpObj;
9         String tmp = (String) obj.get("firstHour");
10
11        long firstHour = Long.parseLong(tmp);//starting point
12        long currentTime;
13        for (int i = 0; i < 24; i++) {
14
15            currentTime = firstHour + i * 3600000;// one hour iteration
16            returnValue[i][0] = currentTime;
17            tmpObj = (JSONObject) obj.get(String.valueOf(currentTime)); // get
    the next Json object
18            returnValue[i][1] = (long) tmpObj.get("lowPrice");          // low
    price of the timestamp
19            returnValue[i][2] = (long) tmpObj.get("highPrice");           //
    high price of the timestamp
20        }
21
22        return returnValue;
23    }
```

Listing 6.13: Reading the Tariff from a JSON String

6.14 Testing Infrastructure

The testing infrastructure is an implementation of the InOutAll interface. This means that all methods used to call the API can be reimplemented to provide the input that is needed for the test.
The InOutAllTest class is an example of this. All methods are returning values that can be set via method and in some cases even construct the wanted objects out of raw data.

An other part of the Testing infrastructure is the implementation ControlTest. It implements the UtilityControl class and provides methods to set the outputs for all inputs.
The last part of the testing infrastructure is the calcMean() method in PricePredictSort. The method is used to calculate the mean value of the past tariff and in order to accomplish this the method performs multiple calls to the API. As this is is not a direct need of the testing infrastructure besides of the mentioned method, the decision was taken to abstract the calcMean() method and add another implementation with a set method.

7 Testing and Results

The chapter describes the methods and assumptions made in order to test the design and proof of concept of the controller. Further the automatic testing done to check the functionality of the methods done via unit tests.

7.1 Implementing a Proof of Concept

The proof of concept is the subject of the majority of the testing process and its structure is described in this section. The full code can be found in the files appended to this thesis, while a shortened version can be found in the appendix under section A.15.
In order to create a new controller some questions have to be answered in order to initialize all parts correctly.

7.1.1 Usage of InOutAll and Settings

InOutAll and Settings are the sole source of information for the system. They should exist only once for all objects.
The InOutAll class is an Interface to enable an easy exchange, if another API has to be used or in this case the inputs and outputs have to be controlled. The Class InOutAllTest implements this functionality. The constructor requires the number of Utilities to be controlled and the number of thermometers as well. With this initialized all requested return values have to be previously set and are in some cases created in the InOutAll class with simple data types as input.
The InOutAllAPIConnect class implements the actual connections to the API via the Interface of InOutAll. Once the constructor is called, the handles and all relevant data is collected from the underlying system and a connection to the API can be established.
The Settings class is not an Interface, as it does not handle complex data connections. All values can be set and read with this implementation and provides optimal requirements for the test.

7.1.2 Usage of Utilities

The implementation of Utilities allows a flexible range of functions, which often include sensors that not every type of utility needs and can be specific to the utility. An example is the Heater. In order to initialize a heater, the number relating to the power plug it is plugged in has to be handed to th e constructor, as well as the sensor number relating to the thermometer that provides the regulation data for the controller of the Heater.
The Boiler is rather low maintenance in this regard, as it only needs the relating number and no connected sensors.
In the proof of concept a third type is used, the TestControl implementation of Utility control allows all inputs and outputs to be controlled externally. This is needed for this test implementation to ensure the correct conditions and to mitigate complex operations concerning the database and thus mitigating the API.
All implementation of UtilityControl that include functionality get the Settings and InOutAll object created for the specific Controller.

7.1.3 Usage of the Optimisation and Sorting Techniques

The class PricePredictOptimisation is an implementation of the Optimisation interface. It provides the scheduler with a method to start optimising. The class does not manage any data other than the List of Sorting techniques that can be included via a set method.
The sorting techniques are handed to the PricePredictionOptimisiation class via Array or List. The sorting technique classes require the InOutAll and Settings classes to be handed to them in the Constructor but don't need any other variables.
One special case is the PricePredictSort class. It has a memeberfunction calcMean(), that provides the mean and standard deviation of the prices. This requires a number of calls to the API and cannot be handled without further complexity on the testing infrastructure. In order to face this challenge the method is set to abstract and the implementation done in class PricePredictSortMean. The testing implementation is provided by the PricePredictSortTest class, which enables the results of the calcMean methods to be set.

7.1.4 Usage of the Scheduler Class

The Scheduler class manages the Utilities and optimisation techniques. It does not require a reference to the InOutAll and Settings class, but receives all Utilities and saves them in the list. Once the optimisation object is handed to the scheduler it is functional.
In order to run the tests, the scheduler only calls the optimiseLists method and the list of Utilities is scheduled in accordance to the sorting techniques. Once this is done another method givePermissions has to be called and all Utilities that are scheduled to get permission are activated, while all without permission are actively deactivated.
For the test, only one call each per set point is needed but in order to run the program in an active environment it has to be properly scheduled and the methods have to be called at the right times.

7.2 Test Cases

In order to fully test the Controller in a consistent way, a full test setup simulating the passing time with sensor inputs and changes in variables like prices, seasons and weather for temperature, consumption of Utilities and the variable consumption of a local home are needed. This is not possible in the scope of the project, as this data is not obtainable at this moment.
The test cases have been chosen to represent usage in different points in a day as described in section 3.3. The numbers are chosen to represent these numbers relatively and in a generalized way.
The assumption is made that nine set points are representative to prove functionality of the scheduler. Further it is assumed that Utility Controller reactions are dependant on time and are replaced by test objects reflecting their probable functionality at these set points. The tests show if the sorting algorithms are sufficient to improve the schedule and achieve the goals of saving money, energy and keeping the comfort of the user.

7.2.1 Set Points

The set points are chosen to represent the most interesting cases of scheduling during the day. Figure 7.16 shows a relative representation of power consumption of a home. It does not represent a time line but illustrates the chosen set points. The following sections explain the reasoning behind choosing the setpoints and explain examples when these can happen during

the cycle of a local home Controller. The black bar represents the consumption while the arrows pointing at it are the setpoints. The text relates the name of the setpoint to the reader. The white arrow in the back is representing the consumption cap.

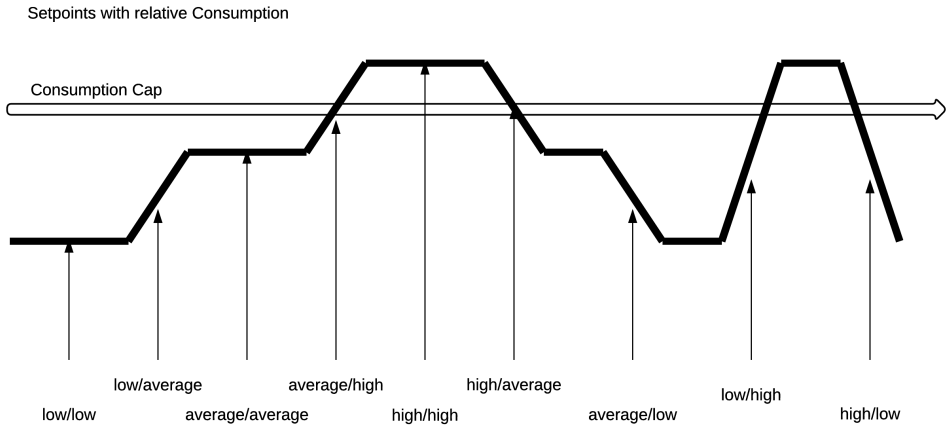

Figure 7.16: Relative Consumption with Setpoints

7.2.2 Setpoint low/low

Low/low refers to a point in the scheduling cycle when the current consumption and price is and has been low and the price in the next hour is low as well. In terms of Utility controllers this refers to a state when priorities should be around 5/10, the temperature is comfortable and the Boiler has had enough time to maintain hot water. The consumption cap is relatively far away and of no concern in this set point.
This set point is representing a point deep in the night. Inactivity within the house means optimal ways to regulate for Utilities; other sources of consumption are off.

7.2.3 Setpoint low/average

Low/Average represents low consumption and priorities around 5/10 with low prices and future prices are average. The Utilities are at an optimal point in terms of comfort and scheduling. Consumption cap is not of concern.
This set point could be found in the morning on a weekend. The usual activities are not concentrated on one point after waking up and are spread over time.

7.2.4 Setpoint average/average

Average/Average represents average consumption and good priorities around 5/10. Price is average and the future price is average. The Consumption cap is of concern, as a local peak could bring the consumption over the cap.

This is an, as the name suggests, average case. A point during the day between lunch and dinner can show these features.

7.2.5 Setpoint average/high

Average/High means average consumption and normal priorities with a comfortable situation at the Utility control. The price at this moment is average; the price in the near future is high. The consumption cap is in immediate reach and has to be monitored
This case applies to the time between afternoon and evening. Power intensive activities like cooking, showering or using electrical devices is in the immediate future.

7.2.6 Setpoint high/high

Consumption was and is high, the priorities of utilities are very likely to be high and the consumption cap is exceeded. The future price is high as well.
A situation like this can happen in the morning during the week or in the evening while cooking etc. Utilities will be activated to provide comfort with high cost.

7.2.7 Setpoing high/average

The end of a peak is reached. Consumption is still high and exceeds the consumption cap, while the Utilities reporting high priorities. The price at the moment is high but will drop to average in the next hour.
This situation is likely to occur in the morning, after breakfast when people start going to work and no extreme consumers like hot plates are active.

7.2.8 Setpoint average/low

Average/Low is referring to a state where the consumption, priorities and prices are average. The consumption cap is of concern but only in consumption spikes and the future prices are low.
The situation can be applied to a summer afternoon, when heating not of concern and people are at work or similar.

7.2.9 Setpoint low/high

The low/high setpoint refers to low consumption now and in the immediate past, priorities are average and prices low. The future prices are high and the consumption cap is approached quite fast.
This set point refers to a busy work day, when cooking; hot water and electrical devices are all used at the same time.

7.2.10 Setpoint high/low

Consumption and prices are high, Utility priorities are high, and the consumption cap has been exceeded. The next prices are low.
Late in the evening, when people go to bed and consumption drops suddenly, is the time referred to in this setpoint.

7.3 Test Setup

An assumption is taken that a test with six utilities are representative of what is expected. This is out of the scope of the testing environment on the Raspberry Pi.
The established set points can be characterized by values in classes InOutAll, Settings and TestControl.

7.3.1 Constant Variables

Table 7.11: Constant Testing Variables

Test Parameters	Class	Value
Consumption Cap	InOutAll	5000
UpdateInterval	Settings	10 min
Mean and Standard derivation of prices	PricePredictSort	20 / 5

Table 7.11 shows the static variables coming from the API, the User and variables calculated through accessing the historical values in the database.
The consumption Cap is set at 5000 units, the update interval is 10 minutes with six updates per hour and a median of 20 units for the price with 5 units as standard deviation.

7.3.2 Utility Parameters

Table 7.12: Utility Settings for Testing

Test Parameters	Heater 1	Heater 2	Heater 3	Heater 4	Heater 5	Boiler 1	Total
Consumption estimation	200	300	150	250	100	100	1100
Priority	1	6	4	7	3	5	

Table 7.12 describes the settings for the Utilities. As mentioned ,six Utilities are used in the test. The Table represents them as Heater one to five and Boiler one.
As per design the two methods provided by the Utilities is the priority and the consumption estimation. The Utilities have consumption estimations from 100 to 300 Units with a total of 1100 Units. The Priorities range from the highest priority to seven, which is in the lowest sector. This covers the spectrum of reactions to be had from the sorting algorithms with an exception of priority ten, which would be completely discarded by design and is not of interest at this moment.

7.3.3 Testing Variables

Table 7.13: Testing Variables - Full Consumption and Tariff

Test Variables	current Tariff low/high price	future Tariff low/high price	full consumption
low / low	10/10	10/10	2000
low / average	10/10	20/30	2000
average / average	20/30	20/30	4000
average / high	20/30	30/50	4000
high / high	30/50	30/50	8000
high / average	30/50	20/30	8000
average / low	20/30	10/10	4000
low / high	10/10	30/50	2000
high / low	30/50	10/10	8000

The variables used in the set points are displayed in table 7.13. The Variables are the prices for the current and the next hour, low and high price each, as well as the consumption of the full system.
The Variables current tariff and full consumption refer to the current hour while the future tariff is seen as the prices in the next hour.
A “low“ hour reflects in a tariff of 10 units as low and 10 units as high price for this hour. This reflects the possibility of both prices being the same value and eliminating the need to work around the consumption cap.
The “low“ value also refers to the full consumption this hour which has the value of 2000.
The “average“ hour has a tariff of 20 and 30 units as prices. This shows the case when both prices are different and the consumption cap optimisation is applied. “Average“ also refers to the full consumption in the first hour, which has the value of 4000 units.
In a “high“ value situation the price is 30 and 50 units. Here the consumption cap is triggered as well and the penalty of consuming more than the cap is higher than in the “average“ case. The “high“ consumption is shown as 8000 and thus already over the consumption cap.

7.3.4 Code Setup

The code used in the Test can be found in the appendix in section A.15. The code shown is the shortened implementation of one setpoint and includes all initialisations needed in order to run the program.

7.4 Test Results

Table 7.14: Test Results

Set Points	Active Utilities	Inactive Utilities	highest Priority	Cap Applied	Energy this Iteration	energy total
low/low	1, 5, 3, 6	2, 4	5	No	241	1450
low/average	1, 5, 3, 6	2, 4	5	No	241	1450
average/average	1, 5, 3, 6	2, 4	5	Yes	575	3450
average/high	1, 5, 3, 6, 2, 4,	-	7	Yes	666	4000
high/high	1	5, 3, 6, 2, 4	3	Yes	1183	7100
high/average	1	5, 3, 6, 2, 4	1	Yes	1183	7100
average/low	1, 5, 3, 6	2, 4	5	Yes	575	3450
low/high	1, 5, 3, 6, 2, 4,	-	7	No	333	2000
high/low	1	5, 3, 6, 2, 4	1	Yes	1183	7100

The results are shown in table 7.14 are ordered by set point as the first column, with the active and inactive utilities after evaluation, the highest Priority used, Information about the Consumption cap usage and about the energy consumed this iteration and during the whole hour, if the schedule stays this way.

7.5 JUnit Testing

JUnit Testing is a simple way of including tests in Java, asserting that a output has a value as expected. JUnit is supported by the used IDE which runs the tests at compile time and reports correct results.
In order to test a feature a new program with desired inputs is generated and the output is checked against the desired result.
Due to the testing framework added, no access to the API is needed. The tests are performed for methods that can be verified by their behaviour. Methods with flexible results depending on previous results are evaluated in different ways.

7.5.1 Example: Activation of an Utility

A Utility represented by a HeaterControl class can be activated by calling method permitUsage(...) with a Boolean TRUE variable. The return value is of type Boolean too and will return TRUE if the Heater has been turned on. In case the Heater was already turned on, a FALSE is returned by permitUsage(...).

```
1 public void testPermitUsage() throws Exception {
2
3         InOutAllTest inout = new InOutAllTest(1, 1);
4
5         Settings sett = new Settings(0,17,23,21,10,5,inout.getCurrentTime());
6         int relaynumer = 1;
```

```
7         int ThermostatNumber = 1;
8         HeaterControl instance = new HeaterControl(relaynumer, ThermostatNumber,
      inout, sett);
9
10        inout.deactivateRelay(relaynumer);  // relay inactive
11        boolean use = true;
12        boolean expResult = true;
13        boolean result = instance.permitUsage(use);// trying to turn it on
14        assertEquals(expResult, result);
15
16    }
```

Listing 7.14: Testing permitUsage(...) in HeaterControl

Listing 7.14 shows the implementation of the test. First the testing environment is created. A new InOutAll object is created with the testing implementation, where all inputs and outputs can be controlled via get and set methods. The InOutAll object defines one relay and one thermostat.
Next a Settings Object, as well as a HeaterControl Object are generated. The HeaterControl object gets the simulated relay and thermostat number as well as the Settings and InOutAll object as inputs.
The preset status is created and simulates the relay to be inactive by calling the appropriate InOutAll method.
The expected result is a Boolean TRUE if the relay is inactive and is turned active. The result is generated by calling permitUsage(...) with TRUE as parameter.
At compile time the assertion in assertEquals(...) is executed. If the two parameters are equal, the test will pass. This concept is applied to multiple different cases and mehtods.

8 Discussion

The discussion provides an overview of the results in relation to the expectations and discusses the usefulness of the project.

8.1 Expected Results

The expectations towards the project were determined by the specifications and further conversations with the supervising parties. The Controller is to schedule multiple types of Utilities as a local embedded application in the context of an e-GOTHAM box.
The focus is on the flexibility and scalability of the design, as the most likely use of the Controller is with different amounts and types of Utilities. Furthermore, the Controller has different aims for scheduling the mentioned Utilities.
The design is created to fit the Controller into an embedded system called e-GOTHAM box. As the e-GOTHAM box is still evolving, an abstraction from the API is desired.
In the context of e-GOTHAM, a customer has a consumption cap specified in the contract. The scheduler should keep the consumption of the whole system, including static energy consumption, beneath said cap, which is applied in an hourly basis.
The concept of a tariff is tied with the consumption cap. The tariff in question provides a 24 hour preview of two prices for every hour. One price for the consumption below the cap, and a second price for consumption above the cap. The scheduler can utilize this to save money by scheduling the Utilities to be less active during the higher prices and be more active depending on the prices in the near future.
The tariff can also be used to balance comfort with cost and consumption. A user can personalize the difference between the two prices and thus apply the cap or leave it out of the scheduling altogether.
Another expectation is for these features are to be implemented and run on a e-GOTHAM testing system.
To summarize the expectations the following list can be applied:

Design features:

- Flexibility to schedule around different variables and Utilities.
- Scalability to include different numbers of Utilities and scheduling algorithms.
- Personalization to tailor the Controller to the individual local home.
- Abstraction of functionality towards the API.

Implementation:

- Different Utility controls for Heater and Boiler.
- Scheduling algorithm to utilize the consumption cap and the tariff.
- Personalization with comfort zone for Heaters and other personal aspects.
- Ability to use abstract calls to communicate with the underlying API.

Proof of concept:

- Simulation of a local home control.

- Usage of multiple Utilities.
- Application of different sorting algorithms and features.

8.2 Resulting Design

The design divides the responsibilities of managing the control of single utilities and the scheduling of the same.
It shows flexibility by allowing different Utilities and sorting algorithms to be implemented. The slim communication between the design entities allow an uncomplicated exchange between utilities and optimisation algorithm and enable the implementation of the two to be exchanged in an simple way, as shown in the test implementation.
The additional abstraction between optimisation and sorting algorithms is added to the design to allow the optimisation to be seen and implemented as one object as seen from the scheduler. This shields the scheduler from the increased complexity of multiple sorting algorithms and keeps the option of the implementation of another form of algorithm.
This abstraction is also part of the scalability factors of the implementation. Any of the Controllers can be added into the system multiple times and are only restricted by hard physical restrictions like number of relays in the e-GOTHAM box or by the system resources like memory and CPU.
The same concept applies to sorting algorithms. If other factors for determining the schedule come into focus, they can be easily added to the class pricePredictionoptimisation or a new algorithm can be impleented as top level optimisation.
The personalization factor of the design relies on the individual controllers and sorting algorithms. The personal information is saved in a different place than the API information and is not deeply interconnected.
The design makes an effort to abstract the API connection towards the classes using them. This is done by creating an extra interface, shielding the API calls. This interface can also implement other ways to access information provided by the API in future implementations or being replaced to enable testing without API and Database.

8.3 Implementation

The implementation reflects the flexibility intent of the design by providing individual controllers for Heater and Boiler, as well as an implementation for scheduling algorithms. The algorithms work with the variables of tariff, the consumption cap and the comfort of the user as input to schedule the utilities accordingly.
The implementation learns from previous data to obtain assumptions about future behaviour. This is shown in the consumption estimation, as well as the price prediction, the cap sorting algorithm and an essential part of the boiler priority generation.
The sorting algorithms and Utilities are passed to the appropriate administrative classes via the construct of a Java List. This limits the scalability to the capabilities of the programming language and the computer system the Controller is run on.
The individual personal variables in the implementation include the lowest, highest and optimal temperatures in a room, the frequency of how often a schedule should be done and an amount of money the customer is willing to pay extra in the tariff to ignore the consumption cap and gain more comfort.
The API calls are abstract behind an interface, and the implementation has the capability to swap all API calls with a testing class that allows individual return values from methods.
A further abstraction has been implemented in the Utility control. It is possible to swap out

Heater and Boiler Controller classes with Test Controllers that are individually configurable. Further parts of the implementation are automatic tests, performed at compile time, to evaluate performance of chosen methods and verify their functionality.

8.4 Test of the Proof of Concept

The tests performed on the Controller run on specific chosen set points and show the effectiveness of the algorithms used to schedule utilities. The set points are assumed to be representative of the average daily activity in a house.
The test show the resulting benefit of running the scheduler with six Utilities and the optimisation algorithm using the consumption cap, future prices and consumption estimations to provide energy and cost savings balanced with comfort for the user.
Accompanied by automatic tests, performed at compile time, to check the behaviour of methods, the performed tests cover most of the design and implementation.

8.5 Test Result Elaboration

The results shown in Table 7.14 can be categorized in multiple groups with the same or similar results.

8.5.1 Result Group One - Mostly Active

Result group one contains the set points “low/low“, “low/average“, “average/average“ and “average/low“. The common factor is the applied priorities. The limit for all of them is at a priority of five, set by the Price Prediction. Further, if the cap is applied or not, is irrelevant, because in all cases the total future consumption is lower than the Cap.
The energy spent per iteration differs between the cases of current average and current low consumption cases, as the prediction for predicted usage relies heavily on the previous consumption. In the case of low consumption we start with consumption per hour of 2000 units and a resulting consumption of 1450, which saves 550 Units or energy. This saving is the same in the cases where an average usage is assumed, with a starting usage of 4000 units and a result of 3450 after scheduling.
In all cases the comfort of the user is not reduced, due to the limitation of priorities at 5.

8.5.2 Result Group Two - All Active

Two set points attract attention due to there being no inactive utilities. The points “low/high“ and “average/high“ both have limiting priorities of 7 and a high future price as common factor. The future price in fact is the factor price prediction sort uses to set the limiting priority to seven.
The result shows a usage of 333 and 666 respectively with no power saving in this iteration. All Utilities are active and consume energy. The Cap is applied in the case of “average/high“, but is not reached.

8.5.3 Result Group Three - High Consumption

High consumption affects the scheduling by shutting everything non-essential down but still using up more than the consumption cap. The Cap is applied and reduces the active utilities

to priority one. Even if the price prediction sets the highest priority to three, as in case of "high/high", the priority three Utility is not active due to the cap.
An energy saving of 900 Units is achieved, with the cost of comfort to the User.

8.5.4 Summary

The results show that the proof of concept does optimisations on the schedule of the Utilities. The benefits range from saving energy and cost in the immediate scheduling period in Group One, storing energy to prevent using high prices in the near future in Group Two and a reduction of consumption to a minimum in Group Three with high prices.

8.6 Interpretation of the Results

The results match the specification, while the implementation has minor inefficiencies. The Controller is working well, but has to be adjusted to the specific needs of a user.

8.7 Result in Context to e-GOTHAM

The result of this thesis can be integrated into e-GOTHAM quite easily, as it already runs on a simulated e-GOTHAM box. Some work needs to be done to adjust the balance between sorting algorithms and utility control algorithms, while the latter need to be evaluated with data.
As the goal of the e-GOTHAM box is to be deployed in local homes, there needs to be more testing done on the Controllers. Finally, a user specific setup process needs to be chosen to ensure the personalization of the final program.

8.8 Limitations

The limitations of the results are mostly due to the testing environment the controller is run in and the needed data for improving the algorithms used. More important the data used in the test is devised by the author and only resembles a real data set. Due to unfortunate timings in the e-GOTHAM development, data directly taken from an e-GOTHAM box was not possible to obtain.
Further considerations are the balancing of priorities with functionality. The design features an approximate way of expressing the current situation and controlling in utilities by choosing an appropriate priority. For the implementation of new utilities or sorting algorithms this basic explanation of priorities and their meaning has to be expanded and adjusted to the program.
The Utility control algorithms implemented for Boiler and Heater are created to represent a generic version of these Utilities. They have to be adjusted to get an optimal performance. This also holds for the prediction methods featured in many of the Utilities and sorting algorithms.
The Utility prediction of usage relies on the mean value of consumption and predicts this consumption for the future as well. In order to predict the static usage of energy the algorithm uses the simple way of subtracting predicted utility consumption from the full consumption.
The e-GOTHAM API as an actor is the continuous contact to the "outside world". There is no direct interaction with the sensors and utilities except through the API. The control over existing data, activation and deactivation of utilities is one major point of failure in the system. The controller does collect some data during interactions to create mean values but the input are all being generated by the e-GOTHAM API in the first place.
Another remark about the e-GOTHAM API is that it uses a local and online database to store values. In the design an important factor is also the speed of the system. In this case the

e-GOTHAM API generates a delay to all interactions of the controller with the system and accuracy and speed of the controlling Algorithm suffers.
The implementation of the design and proof of concept is written in Java and was previous to this project an unknown skill of the author. This affects some of the design decisions, for example the return values or the handling of specific language constructs or parts with dynamic memory.

9 Conclusion

The Conclusion highlights the most important solutions taken from the project and relates it to the specifications and the bigger picture. It is rounded off by adding a personal opinion on the report and adding some limitations that affected the outcome.

9.1 Summary

The final project addresses the issue of lowering the peak load of a local home. This is done by designing a program to schedule and control a local home as part of a Micro Grid. The design provides the tools to schedule Utilities while applying different incentives to optimise the usage. Criteria for the design are flexibility, scalability and personalization of the Controller in regards to the customer.

- The flexibility criteria is fulfilled by the design though offering a straight forward way to introduce new types of Utilities and incentives.
- The scalability is provided by the capability of the design to handle multiple Utilities of the same type.
- A local home Controller demands a certain amount of personalization in regards to Utilities and the gravity of the incentive in comparison to the comfort of the customer.

The design fulfills the aims by splitting the task into controlling individual utilities and scheduling them. The communication between the parts is slim and clear to understand.
A proof of concept implementation is part of the project and implements two types of Utility and employs the e-GOTHAM related incentives of consumption cap and price prediction.
Tests of the design are performed on a Raspberry Pi, simulating an e-GOTHAM box.

9.2 Purpose of the Project

Today's power grids have been built for the electricity usage of the last century and modern requirements to the grid are constantly growing.
One popular solution is the implementation of Smart Grids. They utilize communication and distributed sensor data to schedule loads in an efficient way to prevent failures in the electricity system. To this top level solution a lower level concept is the Micro Grid, in which the e-GOTHAM project can be situated. The approach is to localize part of the load scheduling, while including the benefits of the high level Smart Grid.
The developed Controller is an implementation of a local home control. Incentives for scheduling can be controlled from a higher level and range from a consumption cap, that is fixed in the contract, to an hourly changing tariff that can be changed according to the current or expected load on the grid.
With the proof of concept an incentive and direction is provided to implement other Utilities and optimisation algorithms. The instructions provided by the report are sufficient to include the project into the existing e-GOTHAM box. Once the calibration of specific Utilities and Algorithms

9.3 Resulting Improvements

The design represents a flexible platform that provides the prerequisites to implement optimisation algorithms and Utilities of choice. The tools and instructions on how to use the design

are contained within this report with the example of a proof of concept.
The provided concepts include powerful scheduling algorithms that utilize incentives like the consumption cap and the future price to schedule the example utilities of Heater and Boiler in an efficient way that balances the incentives with the comfort of the customer.
Customisation in form of personal temperature ranges and general settings can be chosen by the customer to personalize the control over the Utilities.
On a higher level, the design and implementation create a bridge between smart Grid control features and localized control to balance the consumption spikes in the daily usage of electricity.

9.4 Personal Opinion

The work on the project was fulfilling and interesting. Working with Geir Mathisen and Sverre Hendseth, as well as Fredrik Bakkevig Haugli and Giancarlo Marafioti was always interesting and gave a good view of the work in a international project of the scale of e-GOTHAM.
The problem this thesis is attempting to solve is complex and fascinating. The effort spent was not wasted and has given me valuable information about my future specialisation in the field. Even though many challenges had to be faced while designing and implementing the project, the result is an important learning experience, academically as well as personally.

9.5 Limitations

The limitations to this project are centered around the testing of the full program. Due to problems in the recovery of the data needed to test the program and its behaviour, the provided tests are minimal. The most effective option of testing the system includes a long term test in a local home or a simulation of home conditions, that react to the controller and simulate the inputs correctly. A full test of all features would require a more thorough and complex testing environment to make sure of all extreme cases, test different setups of combinations between sorting algorithms and Utilities.
Further the provided hardware, in form of a Raspberry Pi, did not provide the full capabilities a e-GOTHAM box is capable of and issues that occur with the extended functionality could not be tested.

10 Further Work

This section relates ideas and improvements that occurred while working on the solution, but are considered out of scope.

10.1 Adding Personalization for Local Needs

In its current state, the Implementation supports the personalization of the Controller in different areas. This includes the temperature in the Heater controller, the frequency of updates for the sorting algorithms and a time frame which specifies how many of the future prices received cannot be changed any more by future versions.
These examples show that personalization is done in all aspects of the Controller, which is enabled through the handling of user specific information through the Settings class.
A proposal is to find more possible options and add them to the implementation. Some examples are as follows:

- creation of time zones: Seasons, Weekend, Weekday, wake up and sleep times, holiday etc.
- Adapt individual settings for the mentioned time zones e.g. Heating lower in summer, longer schedule cycles during the holidays etc.
- Adding these options to an easy accessible User Interface
- Profiles that shift the focus on how much optimization is done without harming comfort e.g. extra power saving mode and comfort mode

10.2 Implementation of Different Utility Controllers

The implemented Controllers represent a small part of the possibilities a local home has to offer in terms of schedulability. Other options like electrical car chargers and air conditioning are prime examples but even time critical appliances like freezers can be scheduled, if a correct controller can be integrated into the system.
Further the current Controllers are not optimal. This can be improved by providing complex mathematical structures and self learning mechanisms to improve the predictions used to schedule the current usage of a Utility.

10.3 Implementation of New Scheduling Algorithms and Sorting algorithms

As stated in the Background chapter, a lot of research has been done in terms of scheduling local appliances. These suggestions can be adapted and implemented into the design to bring efficiency with established algorithms.
In regards to the current algorithms there is still room for improvements as the values for determining the priorities are chosen by what is to be expected, as no data was available to compare the data to.

10.4 Testing in a Realistic Environment

The proof of concept was tested in an environment that does not optimally resemble the conditions of an e-GOTHAM box. This is a point of improvement, where problems with the design can still occur, as experience shows.

10.5 Adjusting the Priority System

The Priorities used as medium of communication between Utilities and optimisation algorithms relies on a scale that reflects the comfort mode of a person affected by the activity or inactivity of the Utility. The priorities can be adjusted to reflect the cases where more room for decisions is needed and where the reasoning is not quite clear to a customer. This has to be tested and researched with concrete targets in mind.

A Test Code

```
1  package egotham.flexibleloadcontrol;
2
3  import egotham.flexibleloadcontrol.Optimisation.PricePredictionOptimisation;
4  import egotham.flexibleloadcontrol.SortingTechnique.CapSort;
5  import egotham.flexibleloadcontrol.SortingTechnique.PricePredictSortTest;
6  import egotham.flexibleloadcontrol.SortingTechnique.SortingTechnique;
7  import egotham.flexibleloadcontrol.control.TestControl;
8  import egotham.flexibleloadcontrol.control.UtilityControl;
9  import egotham.sdi.Clock;
10 import java.util.Arrays;
11
12 /**
13  * Test class for sorting algorithms.
14  *
15  * @author ted
16  */
17 public class Testing {
18
19     /**
20      * Creates a testing environment with TestControl and InOutAllTest objects.
21      * All inputs can be controlled.
22      *
23      * @param setpoint specifies which situation is executed
24      *
25      */
26     public void TestSorting(int setpoint) {
27         InOutAllTest inout = new InOutAllTest(6, 5); // 5 Heater + 1 Boiler, 5
    Thermostats
28         Settings sett = new Settings(0, 17, 23, 21, 10, 5, inout.getCurrentTime
    ());
29         TestControl heater1 = new TestControl(1);       //Initialize Utilities
    and create array
30         TestControl heater2 = new TestControl(2);
31         TestControl heater3 = new TestControl(3);
32         TestControl heater4 = new TestControl(4);
33         TestControl heater5 = new TestControl(5);
34         TestControl boiler1 = new TestControl(6);
35         UtilityControl[] arrU = {heater1, heater2, heater3, heater4, heater5,
    boiler1};
36
37         SortingTechnique cap = new CapSort(inout, sett);    //Initialize sorting
     techniques
38         PricePredictSortTest price = new PricePredictSortTest(inout, sett);
39         SortingTechnique[] arrS = {cap, price};
40
41         PricePredictionOptimisation opt = new PricePredictionOptimisation(); //
    Initialize Optimisation
42
43         opt.setTechniques(Arrays.asList(arrS));         // Add techniques to
    getoptimisation
44
45         Scheduler scheduler = new Scheduler();          // Initialize scheduler
46
47         scheduler.setUtilities(arrU);                   // add Utilities to
    scheduler
48         scheduler.setoptimisation(opt);                 // add Optimisation
    Object to scheduler
49         long[] timestamps = new long[2];
```

```
50          long[] values = new long[2];
51          long[][] tariff = new long[2][3];
52          /*------------------------Initialisation End
    ------------------------------------*/
53          switch (setpoint) {
54              case 1:
55                  System.out.println("");
56                  System.out.println("Case one, low/low");
57                  inout.setConsumptionCap(5000);
58                  inout.settime(Clock.getInstance().currentTimestamp());//time
    doesn't matter
59
60                  timestamps[0] = 1;
61                  timestamps[1] = 2;
62                  values[0] = 0;
63                  values[1] = 2000;
64
65                  inout.setfullConsumption(timestamps, values);   //set
    fullconsumption to 2000
66
67                  tariff[0][0] = 0;
68                  tariff[0][1] = 10;       // low price this hour
69                  tariff[0][2] = 10;       // high price this hour
70                  tariff[1][0] = 0;
71                  tariff[1][1] = 10;       // low price next hour
72                  tariff[1][2] = 10;       // high price next hour
73                  inout.settariff(tariff);                        // set tariff to
     10/10
74
75                  price.setMeanStd(20, 5);
76
77                  heater1.setConsumption(200);//Utilities consumption
78                  heater2.setConsumption(300);
79                  heater3.setConsumption(150);
80                  heater4.setConsumption(250);
81                  heater5.setConsumption(100);
82                  boiler1.setConsumption(100);
83
84                  heater1.setUrgency(1);//Utilities urgencies set
85                  heater2.setUrgency(6);
86                  heater3.setUrgency(4);
87                  heater4.setUrgency(7);
88                  heater5.setUrgency(3);
89                  boiler1.setUrgency(5);
90
91                  break;
92
93          }
94
95          scheduler.optimiseLists();
96
97          scheduler.givePermissions();
98          System.out.println("Test done on setpoint" + String.valueOf(setpoint));
99
100     }
101 }
```

Listing A.15: Implementation of the Test Cases

B Example of Pricing Encoded in a JSON String


```
{"firstHour": "1427212800000", "1427274000000": {"lowPrice": 30, "highPrice":
    30}, "1427238000000": {"lowPrice": 30, "highPrice": 40}, "1427227200000": {"
    lowPrice": 30, "highPrice": 30}, "1427288400000": {"lowPrice": 30, "highPrice
    ": 30}, "1427256000000": {"lowPrice": 30, "highPrice": 40}, "1427284800000":
    {"lowPrice": 30, "highPrice": 30}, "1427212800000": {"lowPrice": 30, "
    highPrice": 30}, "1427263200000": {"lowPrice": 30, "highPrice": 40}, "
    1427270400000": {"lowPrice": 30, "highPrice": 40}, "1427281200000": {"
    lowPrice": 30, "highPrice": 40}, "1427234400000": {"lowPrice": 30, "highPrice
    ": 30}, "1427223600000": {"lowPrice": 30, "highPrice": 40}, "1427295600000":
    {"lowPrice": 30, "highPrice": 40}, "1427252400000": {"lowPrice": 30, "
    highPrice": 30}, "1427241600000": {"lowPrice": 30, "highPrice": 40}, "
    1427299200000": {"lowPrice": 30, "highPrice": 40}, "1427259600000": {"
    lowPrice": 30, "highPrice": 40}, "1427230800000": {"lowPrice": 30, "highPrice
    ": 30}, "1427277600000": {"lowPrice": 30, "highPrice": 40}, "1427220000000":
    {"lowPrice": 30, "highPrice": 40}, "1427216400000": {"lowPrice": 30, "
    highPrice": 40}, "1427245200000": {"lowPrice": 30, "highPrice": 30}, "
    1427292000000": {"lowPrice": 30, "highPrice": 40}, "1427248800000": {"
    lowPrice": 30, "highPrice": 40}, "1427266800000": {"lowPrice": 30, "highPrice
    ": 40}}
```


Listing B.16: Example for a JSON Encoded Tariff String

References

[1] Directive 2009/28/EC of the European Parliament and the council of 23, on the promotion of the use of energy.

[2] e-gotham.eu, last retrieval 07.07.2015.

[3] http://www.nve.no/no/kraftmarked/sluttbrukermarkedet/ams/, last retrieval 07/07/2015.

[4] http://www.nordpoolspot.com/.

[5] Statistics Norway/Statnett.

[6] json.org, last retrieval 07.07.2015.

[7] Definition:microgrid. whatis.techtarget.com/definition/microgrid, last retrieval 07.07.2015.

[8] Ieee guide for smart grid interoperability of energy technology and information technology operation with the electric power system (eps), end-use applications, and loads. *IEEE Std 2030-2011*, pages 1–126, Sept 2011.

[9] IEC/TR, 60725. Consideration of reference impedances and public supply network impedances for use in determining the disturbance characteristics of electrical equipment having a rated current ¿75 A per phase.

[10] Benedicte Langseth Ingrid H. Magnussen Dag Spilde Jun Elin Wiik Toutain Birger Bergesen, Lisa Henden Groth. *Energy consumption 2012, Household energy consumption*. NVE, 2013.

[11] S. Caron and G. Kesidis. Incentive-based energy consumption scheduling algorithms for the smart grid. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 391–396, Oct 2010.

[12] G.T. Costanzo, Guchuan Zhu, M.F. Anjos, and G. Savard. A system architecture for autonomous demand side load management in smart buildings. *Smart Grid, IEEE Transactions on*, 3(4):2157–2165, Dec 2012.

[13] Vera Eckert. *European power grids keep lights on through solar eclipse*. reuters.com, March 2015. last retrieval 07.07.2015.

[14] last retrieval 07.07.2015 Energy.Gov.

[15] T.Hausler G. Mathisen, S.Hendseth. *Master Thesis specification*. NTNU, July 2015.

[16] Gyung-Leen Park Junghoon Lee, Hye-Jin Kim and Mikyung Kang. Energy consumption scheduler for demand response systems in the smart grid. *Journal of information science and engineering*, 27, 2011.

[17] S. Razzaq-M. Ilahi R.D. Khan N. Javaid M.N. Ullah, A. Mahmood. A survey of different residential energy consumption controlling techniques for autonomous dsm in future smart grid communications, 2013.

[18] A.-H. Mohsenian-Rad, V.W.S. Wong, J. Jatskevich, R. Schober, and A. Leon-Garcia. Autonomous demand-side management based on game-theoretic energy consumption scheduling for the future smart grid. *Smart Grid, IEEE Transactions on*, 1(3):320–331, Dec 2010.

[19] Matthew Murray, July 2012. Raspberry Pi review, pcmag.com, last retrieval 07.07.2015.

[20] Kjell Sand Olav B. Fosso, Marta Molinas. *Moving towards the Smart Grid: The Norwegian Case*. Department of Electric Power Engineering, NTNU, 2014.

[21] Alister Doyle reporting by Nerijus Adomaitis and Balazs Koranyi; Editing by John Stonestreet. *Norway wealth fund to ramp up renewable energy investments*. reuters.com, April 2014. last retrieval 07.07.2015.

[22] N. Ruiz, I. Cobelo, and J. Oyarzabal. A direct load control model for virtual power plant management. *Power Systems, IEEE Transactions on*, 24(2):959–966, May 2009.

[23] Jianing Li Suyang Zhou, Zhi Wu and Xiao-Ping Zhang. *Real-time Energy Control Approach for SmartHome Energy Management System*. School of Electrical, Electronic and Computer Engineering, University of Birmingham, Birmingham, UK, 2014.

[24] Qiuwei Wu, Peng Wang, and L. Goel. Direct load control (dlc) considering nodal interrupted energy assessment rate (niear) in restructured power systems. *Power Systems, IEEE Transactions on*, 25(3):1449–1456, Aug 2010.