# Home Automation Systems Integration

*Integrating home automation systems to promote openness and adoption.*

Software Engineering Master Thesis, spring 2010

Tim M. Madsen

**Department of Computer Science**
**Aalborg University**

Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

---

**Title**

Home Automation Systems Integration

**Semester theme**

Programming Technologies and Embedded Systems

**Project term**

SW10, spring 2010

**Project group**

d510b

**Supervisor**

Lone Leth Thomsen

**Co-supervisor**

Arne Skou

**Abstract**

This report documents the development of a range of systems that enable home automation systems to be integrated and exposed to the Web. Requirements for the systems are elicited. Various implementation strategies are considered. The terminology pertaining to the strategies is explored and explained. The strategies are compared, and an informed choice between them is made. Two home automation simulation systems is developed. They are exposed and integrated using the chosen implementation strategy and developed according to the elicited requirements.

**Participant**

Tim M. Madsen

# Preface

The following report is written during the spring 2010 by a single software engineering student at the computer science department at Aalborg University.

When the words *I* or *my* are used, they refer to the author of the report. When the word *you* is used, it refers to whomever is reading this. When the word *we* is used, it refers to you and I.

You are expected to have basic knowledge of object oriented programming and patterns. Other knowledge required is introduced in the report as needed.

When source code is presented, it may differ from actual source. It may have been altered to heighten the legibility the source code. The source code is available online at http://tmadsen.net/sw10-src.zip.

I would like to thank Lone Leth Thomsen for her supervision during this project.

*Tim M. Madsen*

# Contents

# Chapter 1

# Problem Statement

This master's thesis is a project done at Aalborg University in cooperation with its *programming language technology* and *distributed and embedded systems* groups. My preceding three semesters have all been done in this fashion and has allowed for a continuous theme throughout my graduate studies. Below follows a short presentation of my three preceding semesters, they have all involved home automation in on way or another.

**7th semester** had the theme *Internet Development*. The project was about interfacing one specific home automation system, called Innovus, with Web services. Thus allowing information from Web services to control the Innovus home automation system. This was a two person project, described in [CM08].

**8th semester** was about *Distributed and Mobile Software*. An iPhone application that functioned as a mobile and distributed remote control for the Innovus home automation system was developed. In essence, the application enabled multiple users to both manually control devices and set up rules to control devices based on locations on the same home automation system. This was a one person project, described in [Mad09].

**9th semester** was used for a broad investigation in the possibilities of applying rule engines to home automation. Rule engines are quite memory-intensive applications, and the purpose of the project was to reach an assessment to whether it is feasible to run rule engines on an embedded platform used for home automation. This was a two person project, described in [CM10].

The home automation theme continues in this project, that originates from ideas gained through these projects. The rest of this chapter accounts for the problem I will solve in this project and is organized as follows:

**Section 1.1** gives a brief introduction to home automation.

**Section 1.2** discusses what potential benefits home automation offer and some of the problems that still exist in home automation.

**Section 1.3** identifies one problem and state a hypothesis on how to solve the problem.

**Section 1.4** present goals that needs to be reached during an implementation of a solution.

## 1.1 Home Automation Primer

A home automation system is a means that allow users to control electric appliances of varying kind. *Home automation* is also known as *domotics*, a contraction of the words "domestic robotics". When home automation principles are applied to buildings not falling in the "home" category, *building automation system* is a commonly used term.

The most common usage scenario of a home automation system is lighting control, which is fairly easy to both explain and set up. The main components are:

- A hardware controller, or central control unit,
- an actuator, and
- a lamp.

The actuator in this case is a device that controls the flow of current from a wall socket to the lamp in question. It does so by being plugged into both the wall socket, and the lamp. The control unit communicates with the actuator to tell how much current to let through to the lamp. The control unit may be operated through a Web site, a remote control, or something similar. The setup is illustrated in Figure 1.1. The wireless communication between the remote, control unit, and the actuator is done using a home automation communications protocol, e.g. ZigBee [Kin03] or Z-Wave [GEI06].

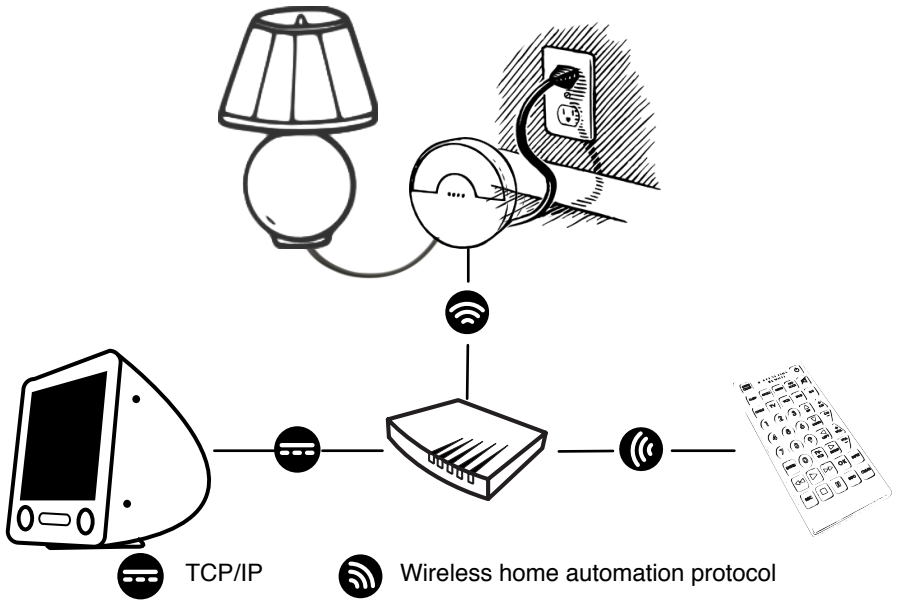


Figure 1.1: A typical home automation setup for controlling a lamp.

## 1.2 Benefits and Obstacles

This section presents the potential benefits from home automation and then looks at some of the obstacles, that are somewhat hindering these benefits.

### 1.2.1 Potential Benefits

The potential benefits we can gain from home automation are almost only limited by imagination and as such it would be infeasible to create a comprehensive list of them. The short list below exemplifies potential benefits in four areas of home automation. The examples are meant to spark the imagination.

**Energy Savings** Through user tracking both in- and outdoors, a home automation system would potentially be able to make sure that, for example, no unnecessary light or heat is turned on in individual rooms.

**Convenience** Trough Web based access to the home automation system a forgetful user will potentially no longer have to worry about if the coffee machine was left on when he left for work. Simply go to a Web page, check it, and turn it off if necessary.

**Security** Tracking user locations can assist in automatic alarm system arming. Also, security cameras might be accessed from a vacation to check that the house is alright.

**Home Entertainment** When engaging in movie watching, the lights might be set to an appropriate dimming level. When listening to music, speakers might be changing from room to room for your listening pleasure throughout the house. Digital paintings on the wall might change according to persons currently occupying the room.

### 1.2.2 Obstacles

For most of the above examples the technology required to realize them already exists: People can be tracked with Bluetooth, RFID chips, or digital people counters, used at some supermarkets and conferences. Some home automation systems feature Web based access to domestic appliances and alarm systems. Other home automation systems come with home entertainment integration, featuring control of television and stereo sets.

With the technology being available, the question is what obstacles are hindering all of the above examples from being common in a setup similar to that of Figure 1.1.

**Proprietorship** Many of the systems (TV, stereo, surveillance camera, etc.) mentioned in the examples are proprietary and as such each have their own programmatic interface that control them, or none at all. Thus to obtain a system able to handle the examples, the buyer has to seek out a home automation vendor that specializes in custom home automation solutions and likely has to buy a whole range of appliances that the vendor is endorsing. This introduces a high cost due to the amount of work required to realize these systems. High cost means that home automation is less likely to become a common household system, unfortunate for both home automation vendors and households.

**Extensibility** Even if the buyer has acquired such a custom system, there is no guarantee that it can be extended with completely new, yet home automation related, features. For instance, the buyer might later purchase a system able to keep track of his refrigerator by means of a camera enabled mobile phone and software able to recognize bar codes. This might be a functionality that the buyer would like to add to his home automation system, much like he would

install a new program on his computer, but most likely will be unable to due to a complicated, or even completely incompatible, system structure.

**Standardization** To obtain an extensible system of home automation related devices, that system must provide an agreed upon standard for device communication. One that companies providing proprietary systems are willing to implement. ZigBee, which is an open source communication protocol, is said to have boosted wireless sensor network standardization [GKC04]. Still, many other sensor network protocols exists and are widely used in home automation solutions, either through wired or wireless communication. Another problem with these kinds of protocols is that they require special hardware to function. In the case with the mobile phone application for keeping track of the refrigerator, a ZigBee, or equivalent, chip might not be available.

## 1.3 Hypothesis

Summing up the problems described in the previous section into one word yields: *communication*.

Domestic appliances such as TVs, stereo systems, lights, heaters, etc. have no standardized common communication platform and consequently it becomes difficult to extend a home automation system to include new features. I hypothesize that it is possible to create:

*A standardized communication platform, that is able to handle communication between many different kinds of home automation related systems.*

As stated in Section 1.1, two common communication protocols are Z-Wave and ZigBee. Section 1.2.2 explains that neither of these are agreed upon standards (at least not in the way as TCP/IP is the agreed upon standard in Internet communication), and how that creates a need for custom solutions that may have a lot of functionality, but still lacks extensibility.

The overall idea for the hypothesized platform is to lift the abstraction level for communicating with domestic devices, thus lift the level for inter-device communication as well as human to device communication. By lifting the abstraction level for communication it is possible to overcome standardization problems with low-level protocols, thus facilitating extensibility and (hopefully) promoting adoption. The major challenge in lifting the abstraction level is to do it in an already standardized way, otherwise the platform will be too hard to use. The way that this project attempts to meet this challenge is described in Chapter 2.

## 1.4 Project Goals

The goals for this project can be divided into the following steps:

1. Define requirements for a system that accommodates my hypothesis.
2. Identify more than one way in which the system can me implemented.
3. Compare the alternatives and make an informed choice between them.

4. Implement a system that accommodates my hypothesis, if possible.
5. Confirm that the implemented system fulfills the defined requirements through testing.
6. Evaluate the system with established requirements in mind and state possible limitations and suggest future work in the area.
7. Conclude upon the project.

Since I am a student, the main goal of any project is to learn new things. During this report I describe many technologies that, before this project, were unfamiliar to me.

## 1.5 Report Roadmap

The rest of the report is structured according to the project goals:

**Chapter 2** identifies the requirements and possible implementation strategies for a system that accommodates the hypothesis stated in Section 1.3. It also elaborates on the vision for the system and states potential benefits.

**Chapter 3** reviews the technical terminology related to the implementation strategies identified in Chapter 2.

**Chapter 4** compares implementation strategies on a conceptual level. The chapter concludes with an informed choice between strategies.

**Chapter 5** reviews the systems implemented during this project to fulfill the hypothesis in Section 1.3. The systems are implemented according to the implementation strategy chosen in Chapter 4.

**Chapter 6** tests the implemented systems according to requirements identified in Chapter 2 to see if they accommodate the hypothesis.

**Chapter 7** concludes upon the project by summarizing the work performed and answer whether the developed systems accommodate the hypothesis.

**Chapter 8** evaluates the shortcoming and future work, on the systems developed and present my thoughts on the technologies used to implement them.

# Chapter 2

# Requirements

This chapter concretize the hypothesis stated in Section 1.3 by first stating a vision for the completed system, a number of potential benefits entailed by that vision, and a scope for this project in Section 2.2. Section 2.3 presents use cases, which serve as implementation and testing guidelines. Lastly, Section 2.4 sum up the use cases in a list of functions that is to be implemented according to use cases. To begin with though, the term "requirements" is defined in the next section.

## 2.1 Requirements Definition

IEEE states in [RGK90] that a requirement can be one of the following:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

The following sections outline requirements as defined in bullet number one, by:

1. Stating the objective of the completed solution.
2. Stating the capabilities needed by users to obtain the objective.
3. Stating the systemic conditions required to obtain the capabilities.

This approach thus includes three levels of requirements. I refer to as them, respectively: *business*, *user*, and *functional* requirements [Wie03]. Figure 2.1 illustrates the flow and products of these individual steps.

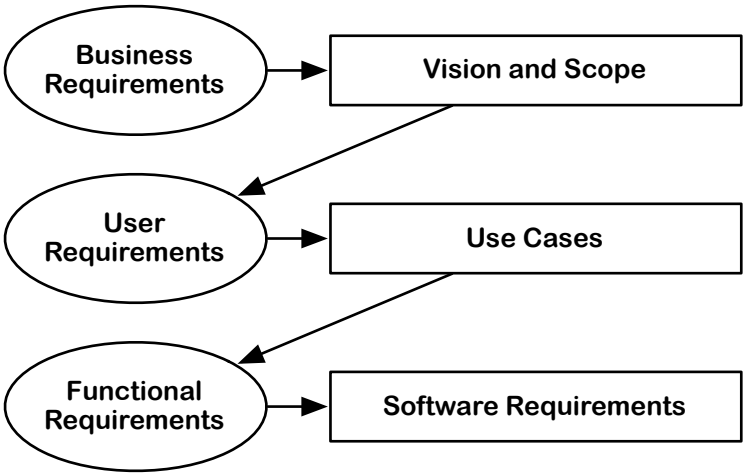


Figure 2.1: A requirement engineering work flow.

## 2.2 Business Requirements

Section 1.2.2 states *proprietorship*, *extensibility*, and *standardization* as some of the current obstacles in home automation. This project aims to alleviate these problems by providing a system that through standardization is proprietor-independent and extensible.

The hypothesis in Section 1.3 states that this is possible by lifting the abstraction level for communication with home automation related devices. A common way of lifting the abstraction level for communication is to implement an interface that acts as a proxy between the home automation related system and the higher level of abstraction. Such a system is also known as middleware [Hoa].

Home automation systems commonly provides a graphical user interface through a browser, and communicates with devices over a wireless network. Due to this networked property of home automation related systems, it is natural that communication with the interface (as suggested by [Hoa] is carried out over a networked structure as well. The Internet is a network that already exists, is highly standardized, and been proven able to handle information sharing in a heterogeneous environment. Thus it is a prime candidate for the higher level of abstraction.

The approach is then to facilitate exposure of domestic devices as Web services or Web resources, encapsulated in either a Service Oriented Architecture (SOA) or a Resource Oriented Architecture (ROA). These terms are reviewed in Chapter 3, which will illustrate differences between the two architectural styles will assist in an informed choice between the two, documented in Chapter 4. For now it suffices to say that the main principle in both SOA and ROA is to utilize the Web as middleware. Web services are Web accessible systems written in an object oriented, functional, scripted, or similar programming environment. The Web service is said to be "bound" to the system in question.

Since this project is about exposing home automation systems as Web services, a suitable working title for the solution is a contraction of the words *Home Automation Bindings*: HAB. The concept of HAB is illustrated in Figure 2.2, which shows a three-

tier architecture that facilitates the exposure of home automation related systems.

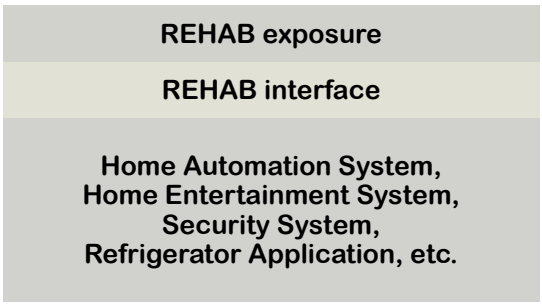


Figure 2.2: The HAB concept. The developer of the home automation related system in question implements the HAB interface that facilitates the exposure of domestic devices as Web services.

There are basically two ways to realize the concept. Approach number one is that the home automation vendors have their own custom application programming interface (API) and a hobbyist developer implements the HAB interface so that it translates messages into the home automation vendor specific API, which, based on personal experience, is often based on XML messages.

Approach number two is that home automation vendors implement the HAB interface themselves. Every vendor has some interface to facilitate control of devices, commonly through a graphical user interface (GUI). Implementing a standardized interface should be in their own interest because the decisions involved in interface development will be obsoleted. Some of the benefits that can be gained by implementing HAB is discussed in the next section.

### 2.2.1 Potential Benefits

Each of the following sections describe a potential benefit triggered by a system such as HAB. The benefits concentrate mostly on home automation systems, but are applicable to any home automation *related* system. Due to the time constraints of this project, these benefits is not considered main goals of HAB, they are food for thought.

#### 2.2.1.1 Aggregation of Home Automation Networks

Some home automation vendors sell systems to large institutions with thousands of devices they want to control from a single location. Unfortunately, there is a limit on how many devices can co-exist in the same home automation network, for instance, the limit for a Z-Wave network is 232 [GEI06]. This means that a home automation system with thousands of devices to control requires a number of control units.

By exposing the devices as standardized Web services it becomes easy for the home automation developers to aggregate their systems in one central system with a user interface that facilitates central control.

#### 2.2.1.2 Out-of-the-box User Interface for Home Automation Developers

With a standardized interface it is easy to represent the devices, and the functions of them on a Web page. This means that home automation developers would no longer have to develop their own graphical representation of the system, they may simply use one provided for them. Cascading Style Sheets (CSS) is a clear-cut way of customizing the look of the Web page(s) to suit the home automation vendor's needs.

#### 2.2.1.3 System Independent Distributed Rule System

A home automation system is inherently based on rules, e.g. when a switch is pressed, light should turn on. This property is easily extendible to more complex scenarios, for instance in the "Energy Savings" example in Section 2.2.1. A standardized interface should allow devices from different systems to subscribe to each others' change in state and act according to a set of user specified rules to achieve desirable behaviour.

Such a rule system would be, in a sense, distributed. Meaning that memory-intensive rule inferencing algorithms such as RETE [For82] [CM10] could possibly be replaced with more simple approaches without a significant change in the time it takes to infer rules.

#### 2.2.1.4 Internet Access to Domestic Devices Behind NAT'ed Networks

Another evident use of HAB would be to implement a feature allowing users on a Network Address Translated (NAT) network to access their domestic devices when on a vacation for instance. The nature of NAT requires some extra work for such a feature to become a reality. HAB could implement this feature once and for all home automation vendors, thus shortening their product development time and making their product more attractive.

### 2.2.2 Scope

The usage scenarios and potential benefits of a finished HAB are manifold, only a few are listed in the previous sections. Unfortunately, a university project can only last for "so long", thus the project needs a scope. The purpose of HAB is first and foremost to show one possible solution to the home automation integrations challenge, using already established standards. This means, for instance, that implementing Internet access to domestic devices behind NAT networks (a potential benefit described earlier) is not a primary objective in developing HAB.

Even though important in real world use, security takes the proverbial back seat when implementing HAB. Security is a big subject, and to say that a system is secure requires *a lot* of testing, which, in turn, requires a lot of time. Still, I will not ignore security either; Safe guards against obvious security holes should be included in any software and they will be in HAB as well. HAB will not, however, make use of secure (encrypted) communications protocols.

Another important issue in real world use is the act of adding new systems to HAB. Ideally, when end users buy a new system, the system should announce itself and

be discovered and added to HAB automatically. Implementing such functionality is not part of this project.

### 2.2.3 Vision Summary

The project vision is to create a standards-based interface that, when implemented by e.g. home automation vendors, enables a central point of control for domestic devices, as illustrated in Figure 2.3. More detailed requirements for fulfilling the vision are described in Sections 2.3 and 2.4. If HAB meets those requirements, it should show that there is an untapped potential in home automation as described in Section 2.2.1.

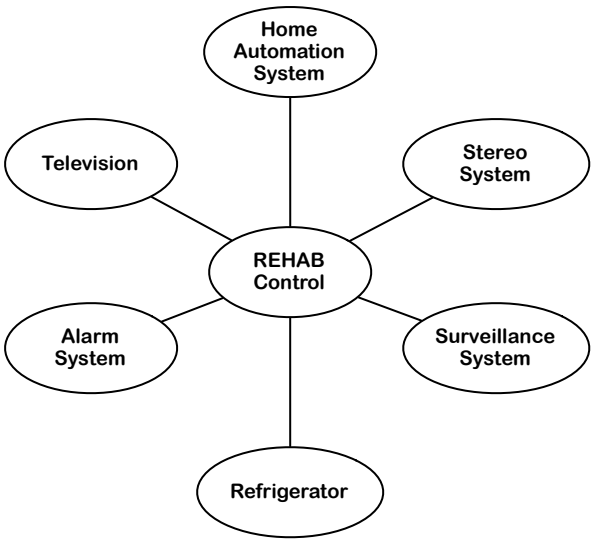


Figure 2.3: The HAB vision, where lines represent communication enabled by the HAB interface.

## 2.3 User Requirements

Exposing domestic devices on the Web implies two kinds of users: interface implementers (as per Figure 2.2) and "Mr. and Mrs. Smith". Implementers are vendors or hobbyists, who implement the interface to enable communication with HAB, and the Smiths are "common folk" who would like to control their domestic devices through a Web browser. Vendors are characterized by having intimate knowledge of the inner workings of their home automation system. Hobbyists are characterized by having access to a home automation system API and know how to utilize it. This section starts of by investigating the needs the Smiths might have, and then goes on to the interface implementers.

### 2.3.1 The Smiths

Each of the following sections has a title, representing a use-case title as suggested in [Wie03]. Each section contains a step-by-step guide to achieving the use-case, and a state transition diagram. Both the step-by-step guides and state diagrams is used during implementation. Each state diagram has a starting point, denoted by a filled black circle, that is the main Web page of the HAB graphical user interface (GUI). The state diagrams also contain a black circle contained within an unfilled circle, representing the end point of the state transitions in a use-case.

#### 2.3.1.1 Add a Device or System

First of all, the Smiths need a means for adding a new system, e.g. home automation system, or device, e.g. the refrigerator application, to HAB. This should be obtained as follows: (illustrated in Figure 2.4)

1. The Smiths navigate to a "System Overview" page on the Web site.
2. They press an "Add System" button.
3. They are prompted to supply an installation file for the system.
4. Once the file is supplied, they press a "Submit" button and one of two things will happen:
   - The file may be erroneous, the Smiths will see the "System Overview" page once again, with a description of the error.
   - The file is accepted, the new system is added, and they will see a page describing the newly added system.

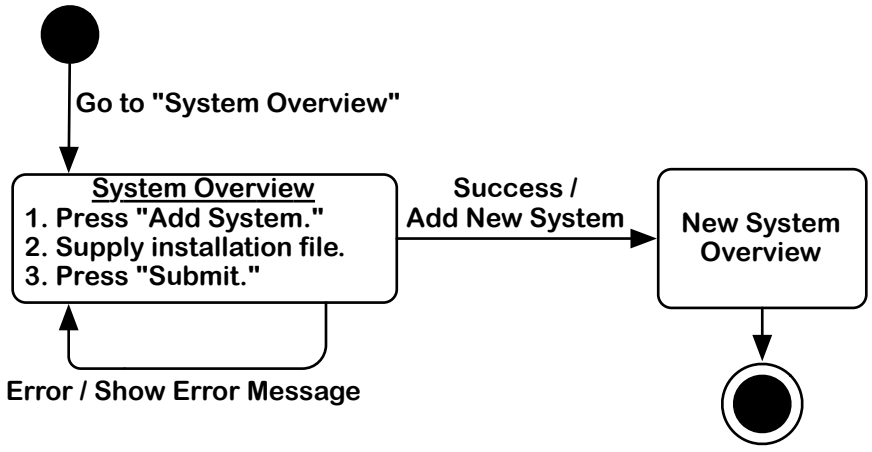


Figure 2.4: The state transitions when the Smiths want to add a new system to HAB.

#### 2.3.1.2 Remove a Device or System

If the Smiths discontinue use of a system already added to HAB they need a means of removing it. Again they use a Web site to navigate to the "System Overview" page.

Here they will press a "Remove System" button, and one of two things happen: (illustrated in Figure 2.5.)

1. If there is no system to be removed, the Smiths will be presented with an appropriate message, and stay on the "System Overview" page.
2. Otherwise, they are presented with a new page, called "Remove System," where they select which system to remove by clicking its name. Upon clicking, they will be asked to confirm the deletion and may choose to either confirm or cancel the deletion.
   (a) Disconfirmation results in being returned to the "Remove System" page without the system being removed.
   (b) Confirmation results in removal of the device, and being returned to the "System Overview" page.

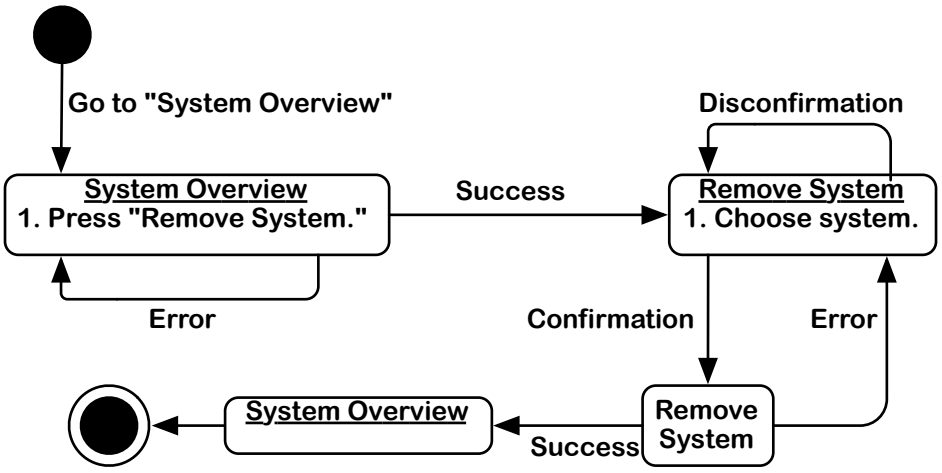


Figure 2.5: The state transitions relevant to removing a system from HAB.

#### 2.3.1.3 See the Status of a Device

When systems have been added, the Smiths may want to see the status of the device(s) included in the system. This is accomplished through a page called "Device Overview," which shows an overview of all device status. If no devices have been added to HAB, the "Device Overview" explains this.

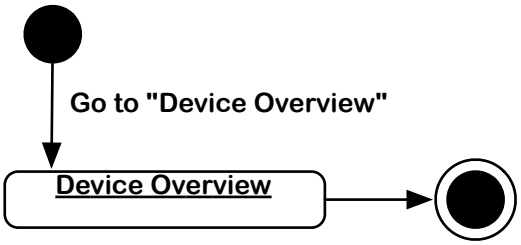


Figure 2.6: The state transition for obtaining an overview of devices status.

#### 2.3.1.4 Change the Status of a Device

The Smiths also want to control devices included in HAB. This can be accomplished either by using the aforementioned "Device Overview" page, or a "Device Details" page. The "Device Overview" page shows information about the current status of a device, but also allows the Smiths to input values to be used for updating the device. Updating a device goes as follows: (illustrated in Figure 2.7)

1. Navigate to either the "Device Overview" or "Device Details" page.
2. Input a new value into the text field associated with the device to be updated.
3. Press an "Update" button, which may yield one of the following:
   - An error, resulting in the same page to be shown again, this time with the relevant error information. The error might be one of the following:
     - If detected at the "Device Overview" page, it is because the input value is unacceptable, e.g. inputting −1 or 101 into a percentage field.
     - If detected while trying to update the device, it is because HAB is unable to communicate with the device in question.
   - A success, resulting in the same page being shown again, this time with the device's updated value(s).

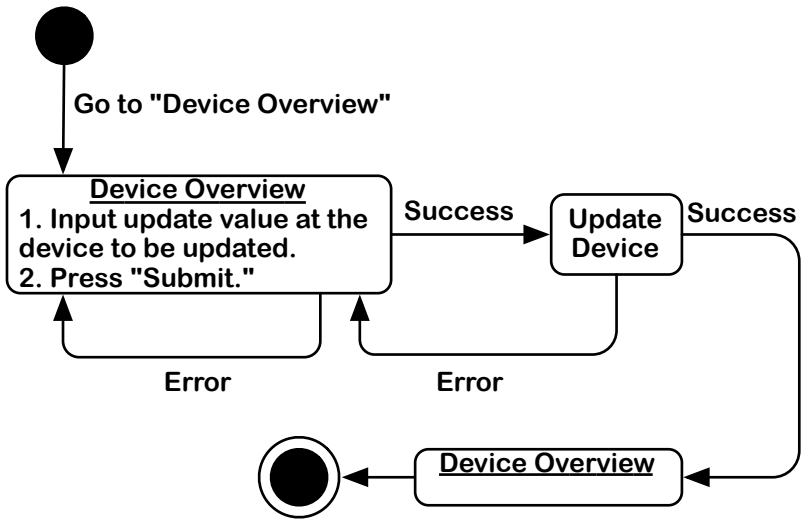


Figure 2.7: The state transitions involved in update a state attribute of a device. The "Device Overview" page can be replaced by a "Device Details" page without difference in state transitions.

#### 2.3.1.5 Create a Rule

Being able to create rules in HAB demonstrates its ability to communicate with different kinds of systems. To exemplify this use-case, it is assumed that the Smiths have added a home automation system that controls their lighting, and a motion sensing system to HAB. Now, they would like to use the motion sensing system to

detect when nobody is at home and, when this is the case they want to turn off lights. Generally, a rule has the form of a control statement known from programming languages:

```
if condition is fulfilled then take action
```

The current example might be formulated as:

```
if motion sensors detect everyone left home then turn off all
                                lights
```

The example can be accomplished by navigating to a page called "Rule Overview" and pressing an "Add Rule" button: (illustrated in Figure 2.8)

1. Configure one or more conditions, by:
   (a) Specifying a device to base the condition on, e.g. motion sensor.
   (b) Specifying a status attribute of that device, e.g. number of people at home.
   (c) Specifying a comparison operator for the status attribute, e.g. "equals."
   (d) Specifying a value to compare against the status attribute by using the operator, e.g. 0.
2. Configure one or more actions to be taken, by:
   (a) Specifying a device to affect, e.g. a lamp.
   (b) Specifying a status attribute of that device, e.g. its on/off state.
   (c) Specifying a value that the status attribute should change to, e.g. "off".
3. Click a "Save Rule" button, which may result in one of the following:
   - An error if the rule cannot be saved. This may happen if the rule conflicts with an already created rule, or if there is a conflict within the rule being created.
   - The rule being saved.

#### 2.3.1.6 Change a Rule

Once created, it might be necessary to change a rule. This is obtained by: (illustrated in Figure 2.9)

1. Navigate to the "Rule Overview" page.
2. Click on the rule to be modified, be taken to a "Rule Details" page.
3. Either:
   - Remove conditions or actions.
   - Add conditions or actions.
   - Modify status attributes of the devices being used in conditions or actions.
4. Click a "Save Rule" button, which may result in one of the following:
   - An error if the rule cannot be saved.
     - Either the comparison values are invalid, or

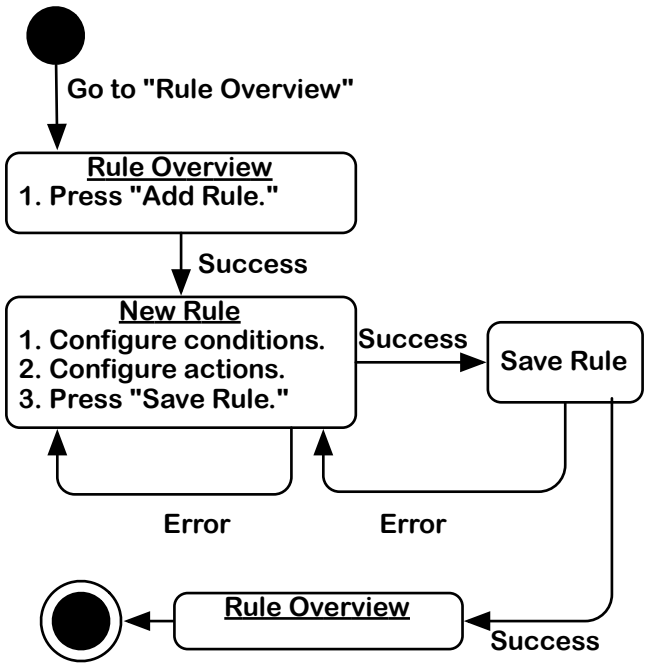


Figure 2.8: The state transitions involved in creating a new rule.

  - the rule conflicts with an already created rule, or
  - there is a conflict within the rule itself.
- The rule being saved.

#### 2.3.1.7 Remove a Rule

Created rules can also be deleted. The Smiths can delete a rule by doing the following: (illustrated in Figure 2.10)

1. Navigate to the "Rule Overview" page.
2. Identify the rule to be deleted, and press its associated "Delete" button.
3. The Smiths are now asked to confirm their decision, which may result in one of the following:
   - Disconfirmation results in being returned to the "Rule Overview" page without the rule being removed.
   - Confirmation results in removal of the rule, and being returned to the "Rule Overview" page.

### 2.3.2 Interface Implementers

The use-cases for the Smiths outline the basic functionality and behaviour of HAB, relevant to achieve the vision presented in Section 2.2.3. The main use-case of inter-

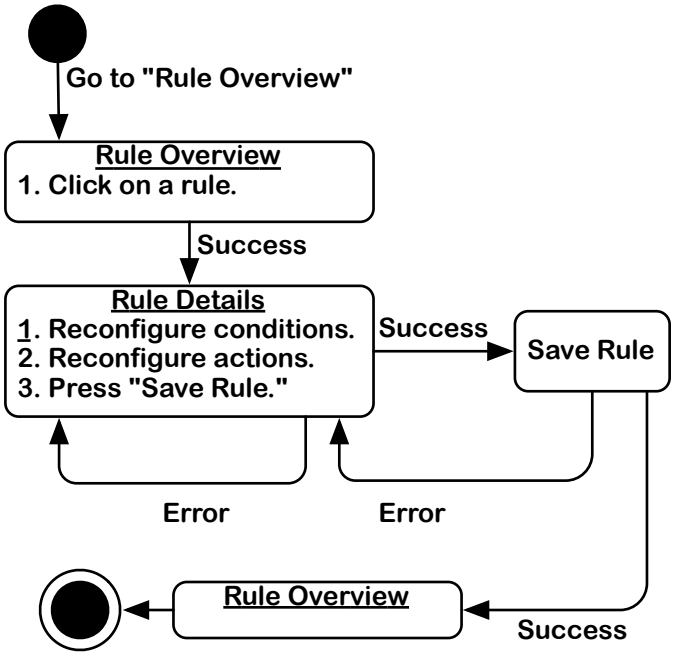


Figure 2.9: The state transitions involved in updating a rule.

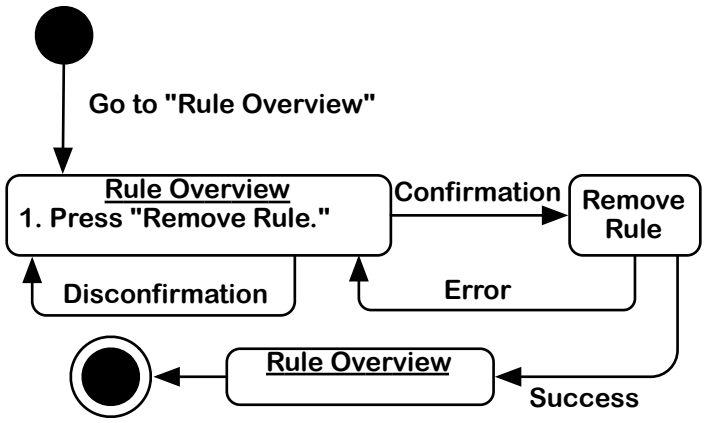


Figure 2.10: The state transitions involved in deleting a rule.

face implementers can be called *Enable Control of System*. The use case is illustrated with a state diagram in Figure 2.11. There is quite a bit of technical know-how required before describing exactly how the communication will be conducted, and this will be devised during the implementation. What can be done now is describe requirements that are desirable for the interface to achieve, which is done in the following sections.

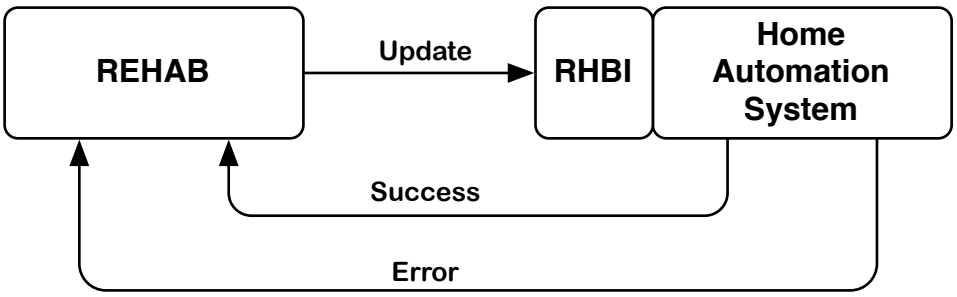


Figure 2.11: HAB issuing an update to a device, through the HAB interface (RHBI). The system containing the device report either success or failure in updating the device.

#### 2.3.2.1 Homogeneity

For HAB to be used by home automation developers, it should be as easy to use as possible. This is obtainable by creating homogeneous interfaces that are easily memorized. While being homogeneous, the HAB interface should also be able to support many different kinds of home automation related systems, i.e. be able to operate in a heterogeneous environment.

#### 2.3.2.2 Convenience

If HAB is to gain a footing in home automation it needs to excel in providing convenience for its users, both for vendor and hobbyist developers. Ideally, it should be as easy to extend the HAB environment with new systems, as it is to install a new application on a computer. This requires a way of describing the system being added, and the devices it contains.

#### 2.3.2.3 Standardization

As mentioned on numerous occasions already, the HAB interface should be based on standards, mainly for two reasons:

1. Established standards are already known in developer communities.
2. Standards are more likely to be long-lasting, than a custom interface developed "in a jiffy."

## 2.4 Functional Requirements

Functional requirements denote the functions that a developer must build into the software to achieve use-cases [Wie03]. The functional requirements for HAB are thus, according to use-cases described in Section 2.3.1:

- Add system.
- Remove system.
- Display device status.
- Change device status.
- Create rule.
- Change rule.
- Remove rule.

These functions will be implemented such that they enable vendor independent system-to-system communication.

The functionality of the HAB interface shall be implemented in such a way that it is homogeneous, based on standards, and convenient to use, according to the requirements stated in Section 2.3.2.

## 2.5 Summary

To quickly sum up, the vision for HAB is to function as a central place from which the Smiths can control their home automation related systems. Furthermore, HAB is a platform for vendor independent system-to-system communication. HAB will be implemented with certain usage scenarios in mind to prove this vision. HAB enables the use-cases through a Web site, which itself is enabled by the HAB interface. The interface will be implemented with desirable properties, specifically *homogeneity*, *standardization*, and *convenience*, in mind. These properties will be demonstrable by the completed system.

# Chapter 3

# Technical Terminology

With HAB being an attempt to expose domestic devices as Web services, we enter a world of buzzwords like: *Web service*, *SOA*, *ROA*, *REST*, *SOAP*, etc. Some of these terms describe technologies while others describe architectures, thus the mixture can quickly become confusing. This chapter provides descriptions of technologies relevant to HAB, covering all of the above terms and other, more basic, terms needed to understand the more elaborate ones.

We start at the very top, by examining what constitutes a Web service, according to authoritative sources on the subject. Then we go all the way to the bottom and take a look at the basic technological commonality of Web service implementations: the Hypertext Transfer Protocol (HTTP). Knowing the basics of HTTP, we are ready to see how HTTP is being utilized in different technologies to obtain Web services of varying architectural styles.

## 3.1 Web Service Definitions

Various authoritative sources have tried to define Web services over the years. This section presents three of these definitions. There are two purposes in presenting three definitions: one is to introduce varying uses of the term, another is to demonstrate that a definitive Web service definition does not exist. The three definitions are presented in each their own section, each section shortly introduces the source.

### 3.1.1 Universal Description, Discovery and Integration Consortium

The Universal Description, Discovery and Integration (UDDI) Consortium was comprised by prominent companies, such as IBM and Microsoft, with the purpose of creating a standard analogous to a "phone book for Web services" [Con00]. Today, the UDDI Consortium is part of another consortium called OASIS; Organization for the Advancement of Structured Information Standards, and the UDDI standard is maintained by OASIS. In 2000, the UDDI Consortium state that:

> *Web services are self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces. Web services communicate directly with other Web services via standards-based technologies.* [Con00]

This is a rather business oriented definition, stating that a Web service is a business application that communicates with other business applications through standards-based technologies. An example usage of Web services that abide by this definition could be a supply chain management (SCM) service. An SCM service can for instance monitor a business's inventory and automatically order parts from another business's SCM service, if this is required.

### 3.1.2 World Wide Web Consortium

The World Wide Web Consortium (W3C) is founded by Tim Berners-Lee (the author of HTML), and maintains standard specifications such as HTML, XML, and SOAP. W3C's definition of Web services, which takes specific technologies into consideration, is from 2004 and states that:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.* [W3C04]

W3C mentions four specific technologies that they consider essential in Web services: WSDL (Section 3.3.1), SOAP (Section 3.3.3), HTTP (Section 3.2), and XML (this report does not dive into the eXtensible Markup Language specification, for more information see [BP+].) With exception of HTTP, these technologies are all W3C "recommendations", which is W3C's word for standard.

### 3.1.3 Richardson and Ruby

Leonard Richardson and Sam Ruby are co-authors on the book "RESTful Web Services" ([RR07].) The two previous sources are strong proponents of the service oriented architecture, while [RR07] explains the principles of a resource oriented architecture. In 2007 Richardson and Ruby simply state that:

> *Web services are Web sites.*[RR07]

Their claim is substantiated through an examination of the process involved in requesting information from both sites and services on the Web. The process is identical in both cases and involves three steps:

1. Find out what you want to request and how you request it.
2. Formulate the request as an HTTP request and send it to the appropriate HTTP server.
3. Parse the response data into data structures that your program needs.

For a Web site user these steps are handled mostly by a Web browser, the user only needs to consider "what to request", e.g. by entering a search term in Google's search field, and click "Search". The underlying Hypertext Markup Language (HTML) of

the Google search site contains the information the browser needs to formulate an appropriate HTTP request and also where to send it. When receiving the response, the browser knows how to parse it in order to display human readable search results. In short, the browser is a program that utilizes services on the Web even though we might think of these services as sites.

## 3.2 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level protocol (on the Open Systems Interconnection (OSI) model [DZ83]). This section is based on [FGM+99].

HTTP is a request/response communication protocol, initiated by a client issuing a request message to a server that sends a response message back. Thus there are two kinds of HTTP messages: *requests* and *responses*. Common for the kinds of messages is that they are composed of a number of *headers* (required) and a *body* (not required). The body contains data relevant to the application to which it is sent, the data can be formatted as HTML (for Web browsers), XML (for XML applications), or any other format — HTTP allows any kind of data in the body. Headers are more strict, here we concentrate only on request headers. They contain information about e.g.:

- Resource location (where a Web site, or service, resides),
- What to do with the resource (e.g. read its contents),
- What content type is being sent (e.g. HTML), and
- What content type is expected in return from the server.

Addressing is obtained through Unique Resource Identifiers (URIs), composed of a host, e.g. www.example.com, and a resource residing on that host, e.g. /index.php. The URI scheme further allows for a query to be performed on the resource denoted with a question mark ("?"). For instance, appending "?page=1&search=xyz" to the index.php resource can indicate to search for "xyz" on page number one, which is delivered by index.php.

Request headers must also specify a *method* (also known as *actions* or *verbs* [RR07]) to be performed on the specified resource. HTTP defines eight methods that can be performed on resources, described below.

**OPTIONS** A request for information about how to communicate with the specified resource.

**GET** Request all information associated with the specified resource.

**HEAD** Request header information associated with the specified resource to check e.g. if the resource is available.

**POST** Request the body of the message to become a new subordinate of the specified resource. This kind of request can result in a new URI addressable resource, an annotation of the specified resource, or an append operation to a database.

**PUT** Request update of the specified resource according to the information provided in the message body.

**DELETE** Request deletion of the specified resource.

**TRACE** A client sending a request with this method name in the header invokes what is called an "application-layer loopback of the request message." This means that by sending a TRACE request, the client requests the server to send back the request message, as received. This can be used for diagnostic purposes, e.g. to examine the chain of servers that have redirected the message from client to server.

**CONNECT** The specification ([FGM+99]) states that this method name is reserved "for use with a proxy that can dynamically switch to being a tunnel (e.g. SSL tunneling [Luo98])." In practice, this means that Internet users that do not have their own IP address, i.e. uses a proxy, can establish a "direct" connection with a server using this method name. The communication between client and server is, in reality, not direct but instead redirected (tunneled) by the proxy, allowing the client and server to establish a TCP connection, that can utilize the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) when transmitting messages.

HTTP is typically used for wrapping documents (much like an envelope) to be transferred between clients and servers. The document contained within the "envelope" can be any number of things, for example HTML, XML, JSON, JPEG, etc. The document is said to have a "content type", which is specified in the "content-type" header of an HTTP message. HTTP requests also specify an "accept" header field, which is a way for the server to return the content-type that the client expects in the response.

## 3.3 Service Oriented Architecture

Service Oriented Architecture (SOA) is an abstraction over a great deal of SOA related technologies. An in-depth explanation of all the technological terms related to SOA would be a project in itself [AKL+06]. Therefore this section superficially explains only essential technological terms.

SOA has come to life through a "strangely competitive and collaborative arena" consisting of software vendors and standards organizations described below [Erl05].

**W3C** As mentioned earlier, W3C is concerned with standardizing Web-related mark-up languages such as HTML and XML, but have also contributed the SOAP protocol (Section 3.3.3) and the XML based WSDL (Section 3.3.1).

**OASIS** An abbreviation for "Organization for the Advancement of Structured Information Standards", whose goal is to promote online trade and commerce via specialized Web services standards. They have contributed UDDI (Section 3.3.2) and ebXML, a set of XML-based standards whose purpose is to provide an open infrastructure for e-businesses.

**WS-I** An abbreviation for Web Services Interoperability is an industry consortium founded by, among others, Microsoft and IBM. The purpose of WS-I is to establish interoperability for selected groups of the Web service standards stack, also known as WS-* (Section 3.3.3.1).

As its name suggests, the basic element in SOA is a service. A service is an abstraction over application logic and business processes. An example of a service could be "create customer order", which may require a number of steps in order to be fulfilled, for example: [Erl05]

1. Retrieve order data.
2. Check if inventory has necessary items.
3. Possibly generate backorder, if some items are missing from the inventory.
4. Generate invoice for the customer.

All of these steps are useful not only in the "create customer order" service, thus each individual step can be provided as a service and be combined into the "create customer order" service. Such a division entails desirable SOA principles such as *reusability* and *composability*. The services know about each other through registration in a service registry, which holds information about how to communicate with each service. The basic principle is illustrated in Figure 3.1 and the following sections each describe one of the technical terms mentioned in the figure. [Erl05]

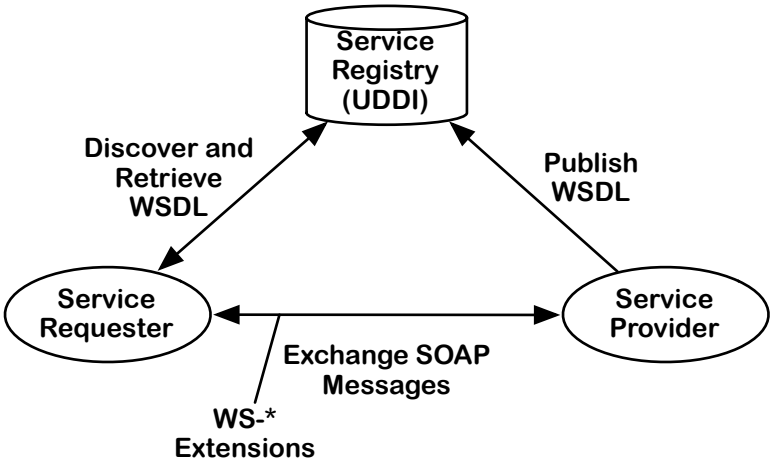


Figure 3.1: The basic principle of SOA.

### 3.3.1 WSDL

The first technology necessary to communicate with Web services is the Web Services Description Language (WSDL), which is a W3C recommendation. WSDL is an XML-based language that provides a component model for describing Web services, the model is illustrated in Figure 3.2. [CMRW07]

**Interface** The component responsible for declaring interface names that a client can use. There may be defined any number (including zero) of interfaces within a description. The *InterfaceFault* component is used to define types of failures that can occur when using the interfaces operations. The *InterfaceOperation* defines operation names, message exchange patterns (Section 3.3.3) and whether or not the operation is safe (regarding side effects), and data types that the interface accepts. The data typing is specified in XML Schema (XSD). If the client does not comply with the data typing, the operation refers to one of the *InterfaceFault* components described earlier. [CMRW07]

**Binding** The interface component defines *what* is to be transferred between service requester and provider, but not *how* — this is the function of bindings. A

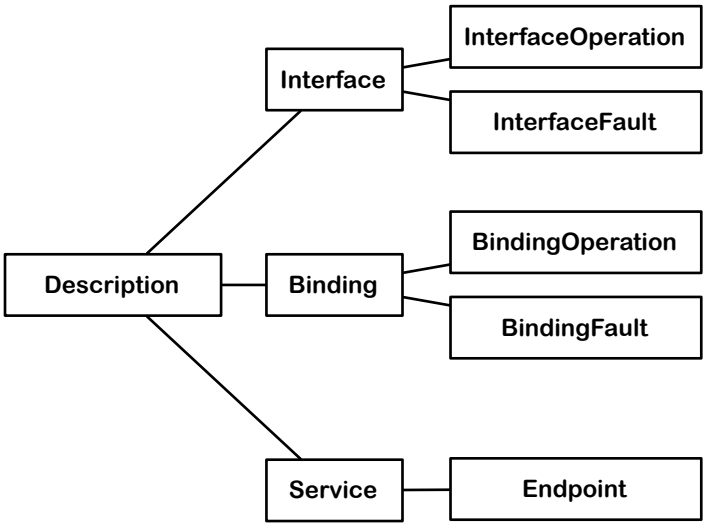


Figure 3.2: The WSDL 2.0 component model.

binding specifies the message format (e.g. SOAP) and transmission protocol (e.g. HTTP) to be used for messages being transferred between requester and provider. "Transmission" is the word used in [CMRW07]. However, a more correct term would be "transfer". Using "transmission" easily leads to confusion with the Transmission Control Protocol (TCP). TCP is in the transport layer in the OSI model [DZ83] while the protocols used for message transfer is in the application layer. Bindings reference individual operations within interfaces so that each operation can use its own message format and transfer protocol (*BindingOperation*). The *BindingFault* is similar to the earlier mentioned *InterfaceFault* in that it will be invoked if the client does not use the correct message format, or transmission protocol. [CMRW07]

**Service** The purpose of the service component is to specify *where* to send messages. A service component references an interface component allowing each defined interface to have their own endpoint. The endpoint component references a binding and further specifies a URI to which messages are to be sent. [CMRW07]

In short, WSDL specifies *what* to send, *how* to send it, and *where* to send it.

### 3.3.2 UDDI

Universal Description, Discovery and Integration (UDDI) is a standard used in service registries to provide a standardized way for humans to look up available services on the Web. There are four data structures involved in a UDDI registry, illustrated in Figure 3.3, providing information which business is providing the service, what the service does, and how to use the service. [CHvRR04]

**businessEntity** Information about the company offering the service, e.g. address, phone number, etc.

**businessService** A description of what the service does, e.g. "Stock Quotes".

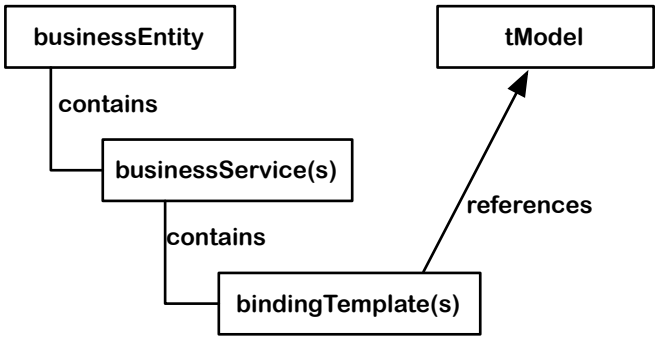


Figure 3.3: UDDI data structures.

**bindingTemplate** Specifies how to access the service, e.g. through HTTP or telephone call. Also provided is a reference to a tModel.

**tModel** Represents the interface that a user, or service, can utilize. Often used to reference the WSDL of a Web service.

### 3.3.3 SOAP

SOAP is a "lightweight protocol intended for exchanging structured information in a decentralized, distributed environment". Formerly, SOAP was an acronym for *Simple Object Access Protocol* — currently, SOAP is a standalone term. [GHM+07]

Messages adhering to SOAP are formatted in XML [GHM+07]. The messages are contained within what is called a "SOAP envelope". A SOAP envelope contains a set of headers and a body. The envelopes can be transferred between client and server using a number of transfer protocols, e.g. the Simple Mail Transfer Protocol[1] (SMTP) or the more commonly used HTTP. Either way a SOAP message looks like the one illustrated in Figure 3.4.

The SOAP body contains XML formatted messages, conforming to the specifications in a WSDL file, if available. The SOAP header can for instance contain information about what encoding the messages use, or whether it is mandatory to parse the header information. The headers also allow extensions to be made to SOAP. One example is the WS-Security extension, which describe how to sign, encrypt, or decrypt a message for instance. [GHM+07] [NKMHB06]

SOAP messages are sent in one of two "Message Exchange Patterns" (MEP), either in a "SOAP Response" pattern, or a "SOAP Request-Response" pattern.

**SOAP Response** A pattern that does not require the client to send a SOAP message to the server. Instead, the client issues an HTTP GET request, and the response contains a SOAP message. [ML03]

**SOAP Request-Response** A pattern that requires the client to send a SOAP message to the server in an HTTP POST request. The service responds with a SOAP message as well. [ML03]

[1] If quibbling over semantics, using SMTP invalidates the term "Web service" as mail is not part of the Web, but rather the Internet.[Kle01]

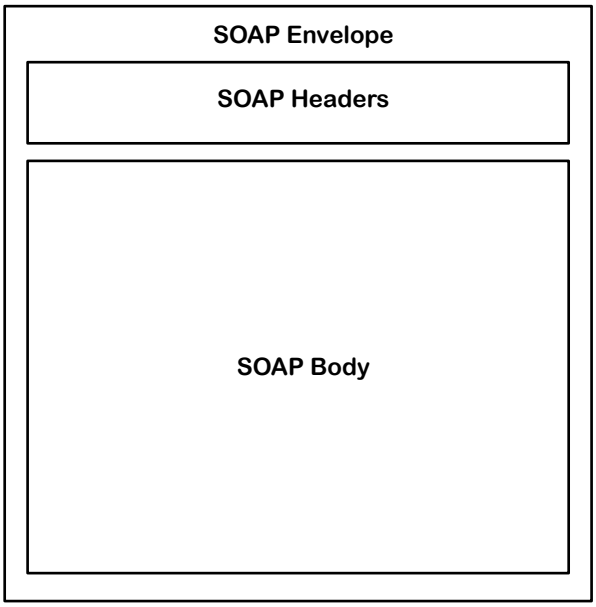


Figure 3.4: A SOAP envelope.

#### 3.3.3.1 WS-*

WS-* is also known as "second generation Web service standards", the first generation standards being represented by WSDL, UDDI, and SOAP. WS-* standards are extensions for SOAP messages that concern security (WS-Security) or addressing (WS-Addressing) for instance. There are also message exchange pattern extensions that provide notification protocols in addition to the two base MEPs mentioned above. [Erl05]

Other message exchange patterns are available through WS-* protocols, but not explained in this report.

The list of WS-* standards is very long (thus not shown or explained here), and not all of them are exactly standards, rather they are protocols under consideration for standardization. Visit `http://www.oasis-open.org/specs` for a comprehensible list of WS-* protocols.

## 3.4 Resource Oriented Architecture

Resource oriented architectures are also known as RESTful architectures [RR07]. RESTful services base themselves on representational state transfer (REST), which is an architectural style described in [Fie00]. REST has guided the design and development of the Web [Fie00]. The basic elements of REST are displayed in Table 3.1 and explained afterwards.

Where the basic element in SOA is services, the basic element in REST is resources [Fie00]. One example of a resource is a *user*. The user resource is a representation of information pertaining to that user, e.g. email address. REST dictates that a

| Data Element | Modern Web Examples |
|---|---|
| resource | the representation of a hypertext reference |
| resource identifier | URL, URN |
| representation | HTML document, JPEG image |
| representation metadata | media type, last-modified time |
| resource metadata | source link, alternates, vary |
| control data | if-modified-since, cache-control |

Table 3.1: REST data elements [Fie00].

resource is addressable with a Uniform Resource Identifier (URI), so a specific user is identified like: *www.example.com/user/1/* for instance. The aforementioned email address that is associated with a user can also be regarded as a resource, identified as *www.example.com/user/1/email* for instance. It is not a strict requirement that each detail about a user is addressable in this manner, the resource granularity is decided by the developer [AA10].

Regarding everything as resources requires a general way of interaction with them. This can be achieved by employing the CRUD (Create, Read, Update, Delete) principle known from relational databases. CRUD conveniently maps to four of the eight HTTP methods:

**POST** For creating a representation of a resource, e.g. a new user.

**GET** For reading a representation of a resource, e.g. an existing user.

**PUT** For updating a representation of a resource, e.g. the user's email address.

**DELETE** For deleting a representation of a resource, e.g. a user who wishes to destroy his account.

The method information lies in the header of an HTTP message, interpreted by the Web server receiving a request. The HTTP body then contains an application specific format, for instance XML or JSON, to be interpreted in context of the method used. Messages sent in RESTful Web service always follows the same "message exchange pattern", request-response. So a deletion request, for instance, can expect a confirmation response that the deletion was actually performed. The response is formatted according to the "representation metadata" data element, which is supplied in the initial request. If relevant, the "resource metadata" element may provide links to alternate representations of the resource in question. The last data element in Table 3.1 is "control data", which can e.g. be used to minimize traffic when dealing with cached resources. A request for a cached resource yields a response with a "Last-Modified" header. A later request for the same resource should include an "If-Modified-Since" header with the value from the "Last-Modified" header. By comparing the two values, the server may respond with only header fields, meaning that the resource has not been modified sine the time specified, or the response will include a body, meaning that the requested resource has been modified since the time specified.

### 3.4.1 Guiding Principles of REST

REST imposes some constraints, or design principles that a Web service must follow to call itself "RESTful". The constraints are described in [Fie00] and briefly summarized

below.

**Client-Server** Implies a separation of concern in that client and server must be separate. The client requests either retrieval or modification of server data through requests and the server informs of success or failure through responses.

**Stateless** The service must be stateless, such that each request from the client contains all the data necessary for the service to understand the intent. If the service was instead stateful, the client could send information that only makes sense in a certain state of the service. That would add a great deal of complexity to the service.

**Cache** To improve network efficiency, responses from the server should be cacheable. By introducing a cache, the service may have to correspond less with system it is exposing, thus providing faster access to data.

**Uniform Interface** By providing a uniform interface, the system exposed by the service is decoupled. REST has four interface constraints: identification of resources, manipulation of resources through representations, self-describing messages, and hypermedia as the engine of application state. The first is accomplished through URIs, the second through message formats such as XML and/or JSON, the third through making the XML and/or JSON messages self-descriptive, and the fourth is accomplished by providing information about to which URIs a client go from the current URI.

**Layered System** A layered systems means that the client cannot see what system lies behind the service. Layers improve scalability by enabling the introduction of load-balancing at the service level. It also further decouples clients from systems, since the two only communicate through a service.

**Code-On-Demand** REST allows clients to download and execute applets or scripts, which simplifies client development by reducing the number of features to be implemented, if these features can be provided by the service.

# Chapter 4

# Choosing an Architecture

We are investigating an easy way to provide home automation integration, we know the requirements for a system that can provide it, and the terminology involved with using the Web as middleware which is how the solution provides the integration. Now, it is time to choose the architectural foundation on which HAB is to be built. As mentioned earlier, there are two alternatives:

- Resource Oriented Architecture (ROA), or
- Service Oriented Architecture (SOA).

Both architectures have strong proponents and opponents, and online debates about which is "best" are long-winded, inconclusive and even resort to name-calling. Fortunately academia take more quantitative approaches to compare the two architectures. This chapter relies on the findings of these comparisons.

This chapter does not try to answer the question of which is best. The truth is that are equally good from an application point of view i.e., both can be used to create systems of equal complexity [RR07]. In this regard comparing ROA and SOA would be like comparing Java and C#, which are both Turing-complete languages, from a functional point of view, which would be a waste of time.

Instead, the two can be compared on other merits like coupling, complexity, and architectural decisions. As for coupling, I briefly present (in Section 4.1) the findings of an article ([PW09]) concerning coupling facets in Web services and how SOA and ROA are either tightly or loosely coupled with regard to those facets. As for complexity, I show (in Section 4.2) how a simple "Hello World" service can be consumed in ROA and SOA environments.

As for architectural decisions, [PW09] compares ROA and SOA from three perspectives:

1. The number of decision that have to be made.
2. The number of alternative options available regarding a decision.
3. The relative cost indicated by development effort required in one architectural style over the other.

The article concludes that less architectural decisions must be made in service oriented architectures. There are more options for each decision because of the many

WS-* protocols. With regard to cost the article states that ROA has a very low barrier for adoption, requires minimal tooling and is thus low-cost and low-risk. However, the article also states that for larger and more complex services it is no simple matter to extend a service built in a resource oriented architecture. This leads to the main conclusion that is to use ROA for "ad hoc" integration over the Web, and to prefer SOA in "professional enterprise application integration".

## 4.1 Coupling

Text books and articles like [PI05] and [Par72] recommend to modularize systems and keep coupling between modules as loose as possible. Modularization and loose coupling is also a defining property of systems implemented as Web services [Kay03], meaning that Web services should be cohesive modules, that can be used with other Web services to form a larger system. The degree of coupling regarding Web services is an expression of how dependent users of the service is on specific details about the service implementation. Tight coupling entails that users (be it clients like you and me or other services) depend on service implementation details. Loose coupling entails that users of a service should be able to use it without knowing anything about how it is implemented.

Coupling in Web services can arise in a number of ways and this section presents 12 coupling aspects based on [PW09]. Table 4.1 summarizes the aspects of Web service coupling, and categorizes ROA and SOA as either loosely coupled, tightly coupled, or neither for aspects that are not clearly dictated by either architectural style.

Each coupling aspect is explained in more detail during the remainder of this section, which finishes off by summarizing the findings of [PW09] and discussing how coupling might have an impact on HAB.

| Aspect | Tight Coupling | Loose Coupling | ROA | SOA |
|---|---|---|---|---|
| Discovery | Registration | Referral | Loose | Tight |
| Identification | Context-based | Global | Loose | Tight |
| Binding | Early | Late | Loose | Loose |
| Platform | Dependent | Independent | Loose | Loose |
| Interaction | Synchronous | Asynchronous | Loose | Loose |
| Interface | Horizontal | Vertical | Loose | Tight |
| Model | Shared model | Self-describing messages | Loose | Loose |
| Granularity | Fine | Coarse | Neither | Neither |
| State | Shared state | Stateless | Loose | Loose |
| Evolution | Breaking | Compatible | Neither | Neither |
| Generated code | Static | None/Dynamic | Loose | Tight |
| Conversation | Explicit | Reflective | Loose | Tight |

Table 4.1: Coupling Facets [PW09].

**Discovery** From Chapter 3 we know that, in SOA, a service requester discovers and retrieves a WSDL for a service through a service registry. ROA, on the other hand is discovered just as ordinary Web sites; by a URI, which may be indexed by a search engine.

A decentralized (referral) means of discovery is more loosely coupled than a centralized (registry) one. Centralized discovery means that for a service to be discovered, another service (the registry) must be available.

**Identification** As described in Chapter 3, entities in a ROA (i.e. resources) is identified universally (globally), through a URI. With a SOA, identification is based on context, meaning that the identity of an entity is only valid within the context of a specific service. Reusing an identifier from one service in the context of another may yield very different results.

**Binding** Refers to resolving symbolic names (e.g. www.example.com) to identifiers (e.g. 192.0.32.10) to be used at a lower level of abstraction. Resolving the server www.example.com to the IP address 192.0.32.10 happens through domain name system (DNS) lookup and is loosely coupled because if necessary, the IP address associated with www.example.com can be changed in the domain name system, without users of www.example.com ever noticing it due to the late binding. Early binding would be the exact opposite, where instead of going to www.example.com a user would have to go to 192.0.32.10.

Due to the extensive use of URIs in ROA, ROA is inherently loosely coupled in this regard. The same is true for SOA; the WSDL for a service defines a URI endpoint and protocol to which the user of a service sends requests.

**Platform** Both ROA and SOA can reside on, and communicate with, heterogeneous hardware and operating systems. Were they instead dependent on, for instance, a specific operating system they would be tightly coupled.

**Interaction** Synchronous interaction means that a service being requested has to be available (online) at the time being requested for the interaction to be successful. The underlying protocol in all ROA and most SOA, HTTP, is often thought as a synchrounous protocol. For instance, if a dynamic Web site's database is unavailable, the site becomes unavailable. This is not entirely true though, as Web sites may be cached and thus delivered even tough e.g. a database is offline. Also, a request that may take a long time to process should yield an HTTP response 202, meaning that the request has been accepted and will be processed. Along with this code, the response should include a URI to a status monitor of the request, or an estimate on when the request has been fully processed [FGM+99]. ROA and SOA is both capable of asynchronous interaction, meaning they are both loosely coupled in this regard.

**Interface** Table 4.1 gives two alternatives for interfaces: vertical or horizontal, which is actually the *orientation* of the interface. Figure 4.1(a) illustrates how a horizontal interface introduces more coupling through an API specifically designed to a service. If the service changes, e.g. to offer new functionality, the API needs to be augmented with this new functionality through new method names, thus the client must be rewritten as well. This is the case with SOA, whose service interfaces are described in WSDL.

The vertical interface (Figure 4.1(b)) shows a client communicating with a service using no API, but only the protocol needed to transfer messages between client and server, e.g. HTTP. HTTP has, as mentioned earlier, eight methods with well-defined semantics as documented in [FGM+99]. Services implemented in ROA implements at least four of the methods, those that corresponds to *create*, *read*, *update*, and *delete* (CRUD.) Adding new functionality in a ROA

service is done through URIs, that can be referenced by hyperlinks. ROA is thus loosely coupled with regard to this aspect.

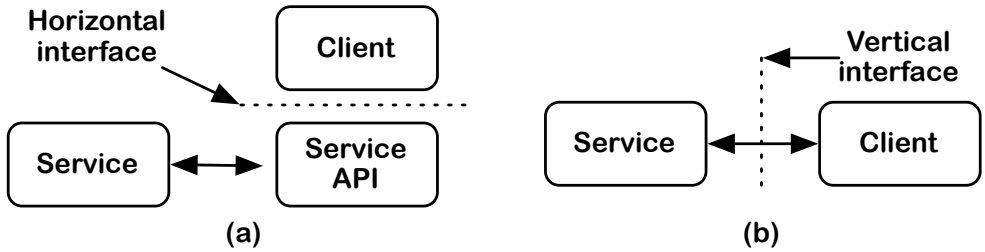


Figure 4.1: Interface orientation [PW09].

**Model** This aspect is about whether the client and service shares data model or not. If the client and service has a shared data model, messages transferred between them may simply be serializations of the model. A shared data model entails tight coupling since the client is intimately aware of the service's inner representation of data. Both ROA and SOA can be implemented in ways that share data model between client and server, but it is not recommended and the practice is usually to transfer self-descriptive messages between client and server, which is the loosely coupled alternative to shared model.

**Granularity** Refers to amount of interactions with a service is required to perform a task. The finer the granularity, the more interactions required. More interactions may put an unnecessary load on the server, that can be avoided by a coarser granularity. Granularity is decided by developers in both ROA and SOA. In SOA, developers decide how many methods their API offer. In ROA, the number of methods is dictated by HTTP, but URI schemes determine the granularity. For instance, http://example.com/lamp/1/brightness refers to a brightness atrribute of "lamp 1", and shows a finer granularity than http://example.com/lamp/1 which just refers to "lamp 1".

**State** Refers to whether the server remembers state or not. An example is a shopping cart service, to which a user incrementally adds items. A stateful service would save the state (items) of the shopping cart either in memory or in a database, but doing so requires lots of interactions (one for each add/delete item, and one for getting the current contents of the cart) and does not scale well. Instead, by implemeting the cart on the client-side (e.g. in JavaScript), the client keeps track of shopping cart state information and the ordering can be carried out in *one* request containing all the items to be purchased. In this last case, the server is stateless, which is preferrable with regards to scaling, but also coupling. Sharing state requires that the client is tightly coupled with the service, so changing the service requires changing the client. If the server instead is stateless, change in either client or service does not require changing the other.

**Evolution** This aspect says something about if evolution (e.g. a new version) of the service breaks clients. As with all other software, this depends on developers and consequently both ROA and SOA can not universally be categorized as either tightly or loosely coupled with regards to evolution.

**Generated code** In SOA WSDLs are often used to generate code, producing tight

coupling between the description and the code. ROA uses declarative mehcanisms, and follow hyperlinks.

**Conversation** In SOA with SOAP, message exchange patterns are defined in both the SOAP specification, but also in some WS-* specifications. The client receives explicit instruction through these specifications on how to converse with the service. ROA takes a more reflective approach because hyperlinks from one resource to another outlines how a conversation can be carried out.

## 4.2 Practical Example

As promised in the introduction to this chapter, this section presents a simple "Hello World" API to a service that simply returns a "Hello `name`" message, where `name` is to be defined by the user of the service. We start by looking at how this is achieved in SOA using WSDL and then in ROA.

## 4.3 SOA Hello World API

The following example is taken from the book "Web Services Essentials" [CL02]. The example exposes a class with one method called `sayHello`. The entire WSDL is shown in Listing 4.1 and explained afterwards.

**Listing 4.1: An API written in WSDL that exposes a "Hello World" service.**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>

   <binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="sayHello">
         <soap:operation soapAction="sayHello"/>
         <input>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
```

```
32                use="encoded"/>
33          </input>
34          <output>
35             <soap:body
36                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
37                namespace="urn:examples:helloservice"
38                use="encoded"/>
39          </output>
40       </operation>
41    </binding>
42
43    <service name="Hello_Service">
44       <documentation>WSDL File for HelloService</documentation>
45       <port binding="tns:Hello_Binding" name="Hello_Port">
46          <soap:address
47             location="http://localhost:8080/soap/servlet/rpcrouter"/>
48       </port>
49    </service>
50 </definitions>
```

Line 2 begins with stating what the service is called, and what namespaces to use. The namespaces help the API programmer later on to reference data types and message exchange patterns.

Lines 9 through 14 defines what kinds of messages are allowed to be transmitted to and from the service. The first `message` tag states that a request to the service must contain one parameter, called `firstName` of type string. The second `message` tag states that as a response the client receives a `greeting`, also of type string. Here we see the first sign of tight coupling; the first `message` tag has a `name` attribute, which corresponds directly to the parameter that the method responsible for returning the greeting takes. If the `name` attribute was `first` instead of `firstName`, the service would not be able to greet you properly.

Lines 16 through 21 defines a `PortType`, which specifies the method to be called in the service class on the server side through the `operation` tag. The `operation` tag further specifies that that the method takes a string as input and returns a string. This is the second example of tight coupling. If the developer wants to change the method name on the server side from `sayHello` to `SayHello` to accommodate a new coding styleguide for instance, the WSDL would also have to modified.

Lines 23 through 41 defines how the messages are to be transmitted and how they are encoded. And lastly, in lines 43 through 49 specifies the location (the URI) of the service.

## 4.4 ROA Hello World API

There is no definitive format, like WSDL, in which to define an API for a services that resides in a resource oriented architecture. What is done instead is asking oneself what the appropriate HTTP verb for a "Hello `name`" request would be. `DELETE` is out of the question, as we are definitely not trying to delete anything. `PUT` seems very unlikely as well, as we are not trying to update any resource. `POST` is not it, because we are not trying to create a new resource in the service. `GET` is the last option then, and makes sense since we are trying to get a greeting from the service. Since `GET` requests means to retrieve a resource identified by the URI [FGM+99], it would be

logical to simply append the `name` to the base URI of the service. So if the service's base URI is `http://localhost:8080/hello/`, a response with a given name would be caused by a request to `http://localhost:8080/hello/name`. This is not something that client developers necessarily have to figure out on their own, the service may present some documentation through its base URI.

## 4.5 General Discussion

Twelve coupling facets from [PW09] have been presented. ROA is clearly more loosely coupled than SOA with regard to these twelve facets. Being loosely coupled is a desirable property in itself and facilitates easier extension of the system. The purpose of HAB is to connect different home automation related systems, thus it is important that HAB facilitates easy extension.

The article presenting concerning architectural decisions ([PZL08]) states that ROA is suitable for "ad hoc" implementations, which is the exact nature of HAB as it should accommodate new types of systems along the way. The conclusion that SOA should be used for "professional enterprise application integration" does not scare me, mainly because the conclusion seems subjective; I find myself asking "Can ROA services not be professional?".

The example though, is the deciding factor. The WSDL API specification is very complicated in my opinion, whereas achieving the same functionality in ROA seems very easy. Thus I use a resource oriented architecture to implement the HAB service. Given that resource oriented architectures build on the principles of REST, it only seems natural to rename HAB to REHAB, for RESTful Home Automation Bindings.

# Chapter 5

# Implementation

This chapter presents the implementation of REHAB. In Chapter 2, the general three tier architecture shows that home automation systems can be exposed to the Web through an interface. This chapter starts by taking a look at two home automation simulation implementations. I have tried getting hold of actual systems that could be exposed, but have not been able to, which is why I instead have implemented simulation systems. After having presented their structure and functionality, I present an interface that enables exposure to the Web. After having exposed the systems to the Web I show the "standardized communication platform" hypothesized in Section 1.3 have been implemented, the system which I call REHAB. Lastly I talk about the experiences with implementing a client for REHAB.

## 5.1 Home Automation System Simulators

The purpose of REHAB as stated in Section 1.3 is to create a communication platform that facilitates communication between home automation related systems. Not having been able to obtain source code for a real world home automation system[1], I have implemented two home automation simulation systems i.e. systems that exhibit the same functionality as real world home automation systems, but without implementing a protocol that communicates with actual devices.

Home automation is briefly described in Section 1.1. This section explores home automation systems further through a method known as "problem domain analysis" [MMMNS00]. After having performed the analysis, I present the implementations of two home automation simulation systems made during this project.

### 5.1.1 Problem Domain Analysis

The two main concepts in a problem domain analysis are: *The problem domain*, which is the part of an environment that is managed, monitored or controlled of a system,

[1]The vendors I have been in contact with uses the Z-Wave wireless protocol, which requires signing a non-disclosure agreement.

and a *model*, which is a description of classes, objects, structures and behavior in a problem domain. [MMMNS00]

We start by looking at the problem domain by considering examples of home automation systems and their use. A common usage scenario for home automation is presented in Section 1.1 and has to do with lighting. The lighting scenario and three other scenarios are described below, they are available for purchase from multiple vendors. The lighting and heating examples described here derive from Danish company flex-control[2], the TV and people counter examples are fictive examples (though controlling a TV through a home automation system is possible). The purpose with the descriptions is to identify the problem domain and later use the observations in creating a model for the problem domain.

**Lighting** A home automation system that controls lighting does so by controlling the level of current let through from a power outlet to the lamp. There are two main ways of achieving this, either by an on/off plug or a dimmer plug. They are both plugged into a power outlet, and the lamp is connected to the plug. The on/off plug has two states as the name implies, either it is on or off. The dimmer plug has the same states but also has a "brightness level" attribute.

**Heating** Heating is controlled through a thermostat fitted with a wireless receiver. The thermostat can be used as any ordinary thermostat by turning the knob to adjust a radiator's temperature, or it can be controlled wirelessly by sending messages that are translated into a thermostat level setting.

**TV** Control of a TV is facilitated through a set-top box. Through the set-top box, the user can control for instance the channel being watched or the volume of the speakers. The set-top box facilitates control through a remote control, similar to ordinary TVs, or through the home automation system's graphical user interface.

**People Counter** In a big building it might be energy efficient to control lighting based on whether there are people in a room. Not by motion sensing, but by keeping a count of how many people are in the room. Such a system would have one or more sensors that each know if there are people in the room in which they are placed.

#### 5.1.1.1 Problem Domain Model

With the above descriptions in mind, we can now make some observations about the common traits in the systems and create a general model for home automation systems.

Following the analysis method of [MMMNS00], we can identify a single class that encompasses all of the above systems (which are objects in the problem domain), the class simply called: *home automation system*. Each home automation system encapsulates a number of *devices* (e.g. dimmer switch, thermostat, or people sensor), that in turn encapsulates a set of *properties* which describe the state of the device. Figure 5.1 depicts a general model for a home automation system.

[2] http://www.flex-control.dk

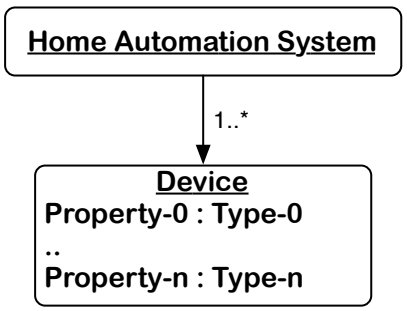


Figure 5.1: A general model for home automation systems. A system has a number of devices, which have a number of properties of varying type.

#### 5.1.1.2 Problem Domain Behavior

The last exercise in the problem domain analysis is describing the domain's behavior. In the examples we see that there are two ways of interacting with a device in a home automation system; either by its "native" control (such as the TV's remote control or the thermostat's knob) or by the home automation system's graphical user interface. If the property change is initiated through the "native control" the change should be reflected in the graphical user interface, and vice versa. The behavioural model is depicted in Figure 5.2.

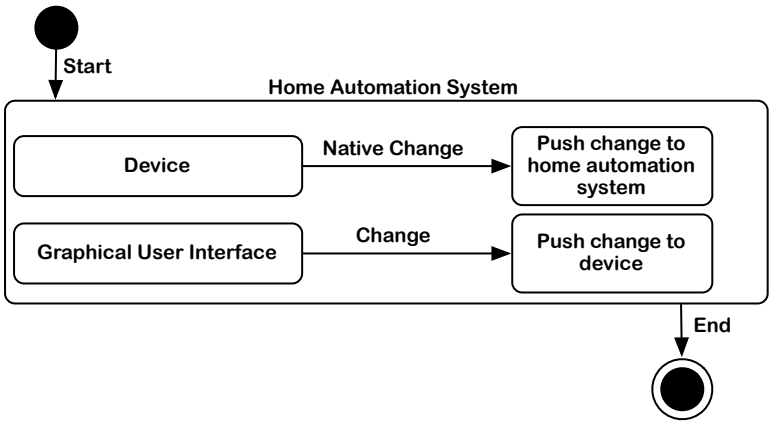


Figure 5.2: The behavioural model for a home automation system.

### 5.1.2 Implementation

With an understanding of the home automation problem domain and its behavior, we are ready to explore the two simulation systems implemented during this project. The two systems simulate a home automation controlled TV and lighting system. I have chosen to implement these two because they are different in that the lighting system has multiple devices, and the TV system only has one device (the TV itself). Since I have been unable to obtain real world source code for a home automation system to see how such systems may be implemented, I have made two assumptions:

1. The protocol used for communication with devices (e.g. Z-Wave or ZigBee) is implemented in C. I find this a reasonable assumption because protocol messages need to be transferred over an antenna, thus a driver is needed. C is a common language for driver implementation due to the low-level operations required on Linux, Mac OS X, and Windows operating systems.
2. The home automation systems themselves are implemented in object oriented languages. I find this a reasonable assumption due to the stateful nature of a home automation system. The state of devices in a system could be polled from the devices every time state is needed, but that would incur unnecessary communication delays.

These assumptions entail some requirements of the language used to implement the simulation systems. The language has to be "friendly" with C and it has to object oriented. The two home automation simulation systems implemented during this project have been implemented in different languages that both meet these requirements.

First, a home automation controlled lighting system has been implemented in Python. Python has a C API, which facilitates extension of the Python interpreter through C programs [Com]. The second system is a home automation controlled TV system implemented in C++. C++ was designed to be "source-and-link compatible with C" [Str], thus C code can be executed from within a C++ program.

Before beginning this project I had no experience with either language. As part of this project's goals is to educate me in new technologies this has been a welcome challenge and the following sections detailing the simulator system implementations highlight not only their functionality, but also what I have found to be notable language features. The following sections present first the lighting-, then the TV simulator.

#### 5.1.2.1 The Lighting Simulator

The lighting simulator is a fairly simple system that pretends to control light switches or dimmers as in the example presented earlier.

We have already seen a generic problem domain model for home automation systems, and the model for the lighting simulator fits in nicely: it has a collection of devices, encapsulated by the `Lights` class. The devices themselves have `LightAppliance` as superclass, which defines a unique identifier and a name for an appliance. `Switch` defines an additional property called `on`, a boolean value indicating if a lamp is on or off. The `Dimmer` class also has the `on` attribute (thus inherits `Switch`) and defines one more property: `level`, an integer describing current brightness level of a lamp. The class diagram is shown in Figure 5.3. Listing 5.1 presents the source code for the `LightAppliance` class.

**Listing 5.1: The LightAppliance class of LightSim**

```
1 class LightAppliance(object):
2     def __init__(self, identifier=-1, name=""):
3         self.identifier = identifier
4         self.name = name
5 
6     @property
7     def identifier(self):
8         return self._identifier
```

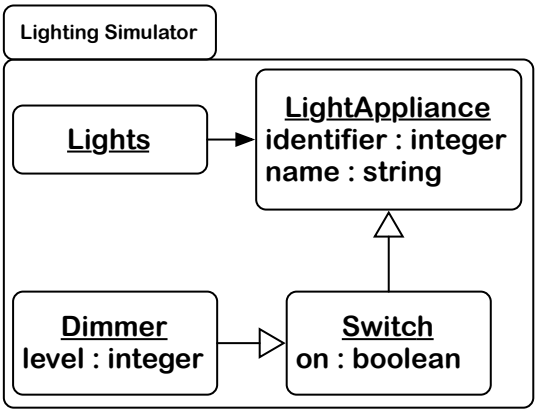


Figure 5.3: The model of the home automation lighting system.

```
9
10      @identifier.setter
11      def identifier(self, value):
12          self._identifier = value
13
14      @property
15      def name(self):
16          return self._name
17
18      @name.setter
19      def name(self, value):
20          self._name = value
```

Regarding the appearance of Python, notice the absence of semicolons and curly braces. Python relies on newline characters for denoting the end of a statement, and indentation (either through tabs or spaces) for grouping of statements, also known as code blocks. Code blocks are initiated with a colon.

If we examine the first line in Listing 5.1 we see the definition of the class itself, and that it inherits `object`, which is the base object for what the Python community calls "new style classes". It is perfectly legal not to inherit anything, in which case the line would simply have been `class LightAppliance:`, but inheriting `object` facilitates the use of property decorators, described shortly.

The second line in Listing 5.1 defines the class initialize method, commonly known as "constructor" in other object oriented languages. Methods in Python may or may not have named parameters, depending on personal taste. In this case, the parameters are named and in addition have default values. In lines 2 and 3, the arguments are passed on property setters defined on lines 10 and 18.

The `@property` decorator syntax in lines 6 and 14 define property getters. For instance `LightAppliance.name` (no parentheses) would return the name of the appliance object, as per line 8. Note that the attribute returned is prefixed with an underscore. This is because there is no notion of private class attributes in Python, everything is public. Instead, "private" attributes are prefixed with an underscore so users of a class know not to get or set that attribute. Lines 10 and 18 demonstrate how to use decorators to define setter methods for properties, that can be used by doing `LightAppliance.name = ''An example''`.

The `Switch` and `Dimmer` classes are very similar, though their inheritance is not of object, but as indicated in Figure 5.3. Their property setters include correctness check of the input, such that a `Switch`'s `on` property can never be anything else than a boolean value and a `Dimmer`'s `level` must be between 0 and 100, both inclusive. As mentioned, the system is only a simulator, but if it was a "live" system, the mentioned property setters should invoked calls to C programs that would relay the changes to the relevant device.

Next up is the `Lights` class in Listing 5.2 that holds references to all instantiated `LightAppliance` objects in the lighting simulator. Normally, I would make sure to make such a class a Singleton (or a Highlander[3]), but while searching on how to implement a Highlander class in Python I stumbled over another design pattern dubbed "Borg". Borg is a reference to the classic science fiction series "Star Trek", where the Borg is a race of cybernetic organisms with the social structure of a bee hive. The important difference between a Highlander and a Borg class is that where a Highlander class is concerned with identity, a Borg class is concerned with state (which is shared among all Borg classes). The important thing is not that there is only one `Lights` object, but that all `Lights` objects hold reference to the same `Switch` and `Dimmer` objects. I implemented the Borg design pattern because it is very easily accomplished in Python and the pattern is presented in Listing 5.2.

**Listing 5.2: The Lights class of LightSim**

```
1  class Lights(object):
2      appliances = []
3      shared_state = {}
4      def __init__(self):
5          self.__dict__ = self.__shared_state # borg design pattern
6          if not self.appliances:
7              self.testFill()
8
9      @property
10     def appliances(self):
11         return self._appliances
12
13     def testFill(self):
14         """
15         Appends test appliances to the 'appliances' property.
16         """
17         switch = Switch(identifier = 0, name = "switch", on = False)
18         self.appliances.append(switch)# add a switch to appliances.
19
20         dimmer = Dimmer(identifier = 1, name = "dimmer", level = 40)
21         self.appliances.append(dimmer) # add a dimmer to appliances.
```

The relevant lines in Listing 5.2 are 2 through 7, where the following happens: First, a class list `_appliances` is defined, then a class dictionary called `__shared_state`. In the initializer method, we leverage the fact that all objects' state in Python is represented through the special `__dict__` attribute; we set that dictionary to our `__shared_state` dictionary. In line 8 we check if the object's `appliance` property has been set, and if not we fill it with some test devices through a method defined on line 13. Now we can be certain that whenever we want a `Lights` object, the appliances it reference have only been instantiated once.

The last detail missing is handling the native controls of the lighting system i.e. when

[3]The tagline of the Higlander movie from 1986 is "There can only be one."

the user presses e.g. a physical light switch. To emulate this behavior, the lighting simulator makes use of the `SocketServer` module in Python's standard library. Upon instantiation of the `Lights` class, a thread is created that handles incoming requests on a predefined TCP port so that changes to appliances may occur from outside the lighting simulator module.

#### 5.1.2.2 The TV Simulator

The TV simulator system has a lot fewer lines than the lighting simulator. This is because the TV is both a system and a device, thus theres is no need for a Borg or Highlander class to reference devices within the TV. The TV is simply a class with, in this case, two properties `channel` and `volume`. Both are represented by integers. The source code is displayed in Listing 5.3 and should be pretty self-explanatory. A namespace called TVSim is defined to avoid cluttering up the global namespace. Then a class called `TV` is defined. Its constructor sets the `channel` and `volume` properties to 4 and 40, respectively. After the constructor, setters and getters for the two properties are defined. The class end with private declarations of the two attributes.

**Listing 5.3: The TV Simulator**

```
1  #include <iostream>
2  #include <string>
3
4  namespace TVSim {
5    class TV {
6      public:
7        TV() {
8          // Initializtion values.
9          this->volume = 40;
10         this->channel = 4;
11       }
12       int getChannel() { return channel; }
13       void setChannel(int channel) {
14         // In "live" system, invoke C code for setting the channel here.
15         this->channel = channel;
16       }
17       int getVolume() { return volume; }
18       void setVolume(int volume) {
19         // In "live" system, invoke C code for setting the volume here.
20         this->volume = volume;
21       }
22     private:
23       int volume;
24       int channel;
25   };
26 }
```

## 5.2 Interface

The two simulator systems described in Section 5.1 do not have any interface available for third parties to interact with. This is, from personal experience, the usual case with home automation systems. As mentioned in Section 1.3 REHAB is an attempt

to provide such an interface to facilitate communication between home automation systems.

From the problem domain analysis in Section 5.1.1 we know that a home automation system consists of devices that have properties. We know from the behavioral model that a home automation system must deal with two ways of interaction with a home automation device, either via the device's native control or via the home automation system's graphical user interface. The exposed interface should thus reflect a system's devices end their properties and allow for users to be notified of any changes in the home automation system. This section deals only with the interface definition and how it is implemented in the simulation systems, Section 5.3 deals with how that interface is used to expose a RESTful interface for third party use.

The interface is defined and implemented in Python. We start by taking a look at the methods the interface define in Section 5.2.1, then how the TV and lighting simulators implement the interface in Section 5.2.2.

### 5.2.1 Interface Methods

The interface in its entirety is shown in Listing 5.5 and defines one initializer method, four methods to be overridden by the home automation system, and two methods that enable third party systems to listen in on changes.

The `get_devices` method should return a list of all the devices in the home automations system, `get_device_with_id` should return a device with the given `device_identifier` parameter, `update_device` enables update of a device property's value from a third party, and `get_metadata` should return the data type and valid inputs on a device's property. Note that after all these method definitions is only a single line with the Python keyword `pass`, meaning that the method is not implemented, only defined, each of these methods are to be overridden by the simulation system (see Section 5.2.2).

The exposure of the interface should not have to concern itself with details about how the devices are implemented in the home automation system, as such concern would require more work than is necessary. Instead all returns are JSON[4] in a format defined by REHAB. The format for a device is a dictionary of its properties and their values and is shown in Listing 5.4. A list of devices (as is required in the `get_devices` method) is achieved by separating devices with a comma, and surrounding the list with square brackets (JSON's array notation).

**Listing 5.4: The JSON device format.**

```
1 {'property_name0': property_value0, 'property_name1': property_value1}
```

**Listing 5.5: The methods in the REHAB interface.**

```
1 import urllib2
2 class Interface(object):
3     def __init__(self):
4         self.listeners = []
5
```

[4]JavaScript Object Notation is a lightweight data-interchange format, comparable to XML. See more at http://json.org

```
6       def get_devices(self):
7           pass
8
9       def get_device_with_id(self, device_identifier):
10          pass
11
12      def update_device(self, device_identifier, property, value):
13          pass
14
15      def get_metadata(self, device_identifier, property):
16          pass
17
18      def register_listener(self, listener_url):
19          self.listeners.append(listener_url)
20
21      def device_was_updated(self, device_identifier):
22          if self.listeners:
23              for listener in self.listeners:
24                  urllib2.urlopen(listener).read()
```

The last two method in the interface are already implemented and enables third parties to listen on changes made in the home automation system. This is achieved enabling third parties to register a URI to be called whenever a change happens. The URI should be bound to a controller that will fetch the changes from the home automation system. It could have been implemented so that the changes be sent in the body of the HTTP request issued, but that would incur some inconvenience. First, the third party should maintain state of devices on their side (and update the appropriate device's property with the new value), which should not be their concern as they should be oblivious to device implementation. Second, it complicates refactoring in both the home automation system and the third party system if for some reason, the JSON format of devices would be changed for some reason.

### 5.2.2 Interface Implementation

Since the main contribution of this project is to provide an easily implemented interface, that enables effortless exposure to third parties, I will go through the implementation of the interface in both the lighting- and TV simulation systems in the following sections. The two simulation systems' implementation of the interface are tightly coupled to the systems' individual implementation. This is only natural, but the interface provides a homogeneous representation of the systems, which is leveraged when exposing the systems to third parties in a loosely coupled manner. Exposure of the systems is explained in Section 5.3.

#### 5.2.2.1 Lighting Simulator

The lighting system is already written in Python, so all that has to be done when implementing the interface is three things:

1. Import the interface module (called `RHBI`).
2. Define a class that inherits the interface class presented in Section 5.2.1.
3. Override the four system specific methods presented in Section 5.3

One thing still remains; since the "get" methods in the interface should return JSON formatted devices and meta information on properties we need a way to encode `LightAppliance` objects (Section 5.1.2.1). Python's standard library has a module called `json`, which is able to turn standard Python types into a JSON object. This means all the simulator needs to do is convert `LightApplicance` into a dictionary of its properties (as per Listing 5.4), which in turn can be encoded in JSON, effortlessly.

Converting a `LightAppliance` into a dictionary a fairly easily accomplished task as shown in Listing 5.6.

**Listing 5.6: The lighting simulator's REHAB encoder.**

```
def rehab_encode(self, la):
    # la means LightAppliance
    properties = [n for n in dir(la) if not n.startswith('_')]
    attributes = {}
    for property in properties:
        attributes[property] = getattr(la, property)

    return attributes
```

The method in the above listing takes a `LightAppliance` object and creates a list of its properties using a list comprehension and a Python built-in function called `dir()`. The `dir()` function takes an object as input and returns a list of the object's attributes. Since it is conventional in Python to prefix private attributes with an underscore we only want those not prefixed with an underscore i.e. public attributes, in this case, the properties defined on a `LightAppliance`. For instance, the property list for a `Switch` object would be `['identifier', 'name', 'on']`.

After having created a list of properties we use another Python built-in function called `getattr(x, 'attr')`, where `'attr'` is a string of the named attribute we want to access on `x`, to create a dictionary representation of the given `LightAppliance`.

Now we are ready to examine the interface implementation in Listing 5.7. It's a relatively simple matter now that a utility method convert an appliance into a built-in data type. The `get_devices` method simply iterates the appliances property of the `Lights` Borg class, encodes each appliance as a dictionary, appends the dictionary to a list and returns the list as a JSON object. `get_device_with_id` is similar but only returns one appliance as a JSON object.

**Listing 5.7: The lighting simulator's REHAB interface implementation.**

```
import json
import RHBI # The rehab interface.
class RHBInterface(RHBI.Interface):
    # You need to define the string that serves as key for the device
    # identifier here. This is used when exposing the interface.
    identifier = 'identifier'

    def get_devices(self):
        devlist = []
        for appliance in Lights().appliances:
            devlist.append(rehab_encode(appliance))

        return json.dumps(devlist)

    def get_device_with_id(self, device_identifier):
        for appliance in Lights().appliances:
            if appliance.identifier == device_identifier:
```

```
18                 return json.dumps(rehab_encode(
19                     Lights().appliances[device_identifier]))
20         return None
21
22     def update_device(self, device_identifier, property, value):
23         for appliance in Lights().appliances:
24             if appliance.identifier == string.atoi(device_identifier):
25                 dev = appliance
26
27         if dev:
28             setattr(dev, property, value)
29
30     def get_metadata(self, device_identifier, property):
31         meta = {}
32         if property == "on":
33             meta['type'] = 'bool'
34             meta['values'] = {'false':'off', 'true':'on'}
35             return json.dumps(meta)
36         elif property == "level":
37             meta['type'] = 'int'
38             meta['values'] = {'min': 0, 'max': 100}
39             return json.dumps(meta)
40
41         return None
```

The `update_device` method also iterates all the appliances, finds the one with the relevant identifier, and then uses Python's built-in function `setattr` to update the relevant attribute through the property's setter method.

The last overridden method in the interface returns meta data on a property, i.e. the property's type and possibly valid values. The meta data is used when representing the devices in a graphical user interface, where the meta data is translated into interactive controls on the screen. The `get_metadata` method in Listing 5.7 is not very elegant, the meta data should reside in the relevant `LightAppliance` classes.

#### 5.2.2.2 TV Simulator

Since the TV simulator is implemented in C++ and the REHAB interface is implemented in Python, the TV simulator needs to be made available to the Python interpreter in order to implement the interface. To that end, the Boost.Python[5] library has been used. Boost.Python enables C++ structures (such as classes end methods) to be compiled into a Python module.

It works by defining the classes and methods to be available to Python. Once defined the C++ file is built using Boost's make system called Jam. Jam produces a shared object (`.so`) file that can be imported in Python as if it were a Python module. Listing 5.8 shows the source code that defines which classes and methods to make available in the resulting shared object, the code is included in the same class as the TV class from Listing 5.3.

**Listing 5.8: Making the TV simulator available as a Python module using Boost.Python.**

```
1 #include <boost/python.hpp>
2 using namespace boost::python;
3
```

[5]Boost is a collection of libraries for the C++ programming language.

```
4 BOOST_PYTHON_MODULE(TVSim)
5 {
6   class_<TV>("TV", init<>())
7     .def("setChannel", &TV::setChannel)
8     .def("getChannel", &TV::getChannel)
9     .def("setVolume", &TV::setVolume)
10    .def("getVolume", &TV::getVolume)
11  ;
12 }
```

Since there is only one TV (unlike `LightAppliance` objects in the lighting simulator), both the encoder and interface looks a little different from that of the lighting simulation. Let us take a look at the encoder first, in Listing 5.9.

**Listing 5.9: The TV simulator's REHAB encoder.**

```
1 def rehab_encode(self, obj):
2     attributes = {}
3     attributes['channel'] = obj.getChannel()
4     attributes['volume'] = obj.getVolume()
5     attributes['identifier'] = 1
6     attributes['name'] = "TV"
7     return attributes
```

As in Listing 5.6, the TV object is translated into a dictionary representation. The difference is that since there is only one TV object, it is sufficient to "hard-code" the attribute keys instead of iterating over a list of properties.

The interface implementation also differs because the TV system only has one appliance, see Listing 5.10. Whenever the variable `tv` is used in Listing 5.10 it is referencing a global (in a module in which the interface is implemented) `TV` variable. The `get_devices` method simply returns a list with only one element, the `get_device_with_id` disregards the `device_identifier` argument and returns the JSON encoded TV object. The `update_device` method also disregards the `device_identifier` argument and updates the specified property with to the provided value.

**Listing 5.10: The TV simulator's REHAB interface implementation.**

```
1 import json
2 class RHBInterface(RHBI.Interface):
3     identifier = 'identifier'
4
5     def get_devices(self):
6         return json.dumps([rehab_encode(tv)])
7
8     def get_device_with_id(self, device_identifier):
9         return json.dumps(rehab_encode(tv))
10
11    def update_device(self, device_identifier, property, value):
12        if property == 'channel':
13            tv.setChannel(string.atoi(value))
14        elif property == 'volume':
15            tv.setVolume(string.atoi(value))
16
17    def get_metadata(self, device_identifier, property):
18        meta = {}
19        if property == "volume" or property == "channel":
20            meta['type'] = 'int'
21            meta['values'] = {'min': 0, 'max': 100}
22            return json.dumps(meta)
```

```
23
24          return None
```

## 5.3 Exposure

With an implemented interface, we are ready to look at how the home automation systems are exposed. Before we do, we follow up on interface orientation introduced in Section 4.1.

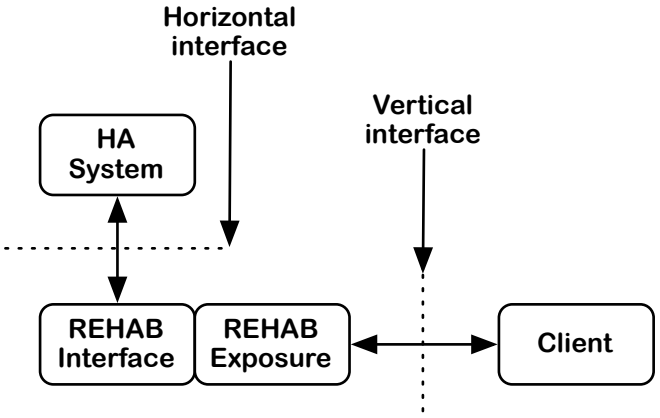


Figure 5.4: The interface orientations relevant to the interface and exposure of a home automation system through that interface.

As can be seen in Figure 5.4 the interface orientation between the home automation system and the REHAB interface is horizontal, i.e. the home automation system and interface is tightly coupled, also mentioned in Section 5.2. The goal is to provide the client with a vertical interface, i.e. provide an interface to the client which is loosely coupled (the client need not know anything about any implementation details). The vertical interface is achieved by exposing information (obtained through the interface) in a RESTful webservice.

### 5.3.1 Web Framework

The RESTful exposure service has been implemented using a Web service framework for Python. There are multiple options when it comes to Python Web frameworks, the most commonly known is probably Django[6] and Pylons[7].

Having no previous experience with either framework I took a quick look at them (both the frameworks' come with a "Getting started tutorial"), and found them both to be quite similar to the Ruby on Rails[8] Web framework for Ruby. I have previous experience with the Rails framework and while it is very popular among many Ruby

[6] http://djangoproject.com
[7] http://pylonshq.com
[8] http://rubyonrails.com

developers, I found it to be too complex for my taste, too much "framework magic" happening, that I didn't grasp.

Not wanting to face a framework similar to Rails again, I continued my search on Pythons wiki page on web frameworks[9] and stumbled upon one called web.py[10]. It seemed very minimalistic and very easy to use. As an example, the first thing that meets the eye on the website is a complete Web application in 10 lines of code that is easy to understand. After having seen that incredible simple example and after having written a RESTful blog application to explore the framework a little further, I decided to write all the Web services in web.py.

### 5.3.2 Exposure Implementation

In this section you will see how a RESTful Web service has been written in Python, using the web.py framework. Almost all the source code is presented in this section.

The first thing to decide is how to identify resources through URIs. Considering the findings in the problem domain analysis this is fairly simple. We know that a system has devices that in turn have properties. Thus, the URI scheme in Listing 5.11 seems logical.

**Listing 5.11: The URI scheme exposing a home automation system.**

```
1 uris = (
2         '/', 'Index',
3         '/listeners', 'Listener',
4         '/(.*)', 'Device',
5         '/(.*)/(.*)', 'Property',
6         '/(.*)/(.*)/meta', 'PropertyMeta'
7         )
```

In line 1, a tuple called `uris` is defined. Odd-numbered elements in the tuple are URIs with support for regular expressions, `(.*)` means zero or more characters of any kind. The even-numbered elements in the tuple defines class names that should handle requests made to URIs matching the preceding elements. The URI scheme in Listing 5.11 fulfills the REST principle called "Identification of resources". In Listing 5.12 you can see how the `Index` class handles requests to `/`.

#### 5.3.2.1 Index Class

**Listing 5.12: The `Index` class in the exposure service.**

```
1 class Index:
2     def GET(self):
3         devices = json.loads(interface.get_devices())
4         uris = []
5         for dev in devices:
6             uristring = '{0}/{1}'.format(
7                     web.ctx.home,
8                     dev[interface.identifier])
9             uris.append(uristring)
10
```

[9] http://wiki.python.org/moin/WebFrameworks
[10] http://webpy.org

```
11          return json.dumps(uris)
```

Line 1 defines the class, line 2 defines a method named `GET`, which is called whenever an HTTP `GET` request is issued to the index URI. This is where I find web.py extremely easy to use, each class may define as many methods as desired, but the ones named after the HTTP methods is called when such a request is issued. The purpose of the index URI is to give the user an overview over which devices are exposed through the service.

Line 3 fetches the array of devices through the interface. The `interface` variable is defined by the home automation developer, and is the only variable that differs in the exposure services for the lighting- and TV simulation systems. After having fetched the devices, an iteration over them is performed to create a list of URIs identifying device resources. The `web.ctx.home` variable in line 7 is a variable within the web.py framework that returns the protocol used to make the request and the application's base path, for instance `http://localhost:5050`. In line 8 the identifier variable in the interface (see Listing 5.10) is used make the URI unique, for instance a device with `0` as its identifier has URI `http://localhost:5050/0`. The last line returns the list of URIs in JSON notation, exemplified in Listing 5.13. Ideally, the service should be able to return XML and HTML responses as well, but unfortunately time did not permit it. Returning different representations of resources (such as the one in Listing 5.13) should be done by looking at the `ACCEPT` header in the client's HTTP request and if the header states that the client accepts `application/xml`, an XML encoder should be used in place of the JSON encoder.

**Listing 5.13: An example of the JSON response to a GET request on '/'.**

```
1 ["http://localhost:5050/0", "http://localhost:5050/1"]
```

This example shows how all the REST principles from Section 3.4.1 is accomplished. The index resource is a list of URIs to device resource, showing that application state is driven by hypertext. The uniform HTTP interface is available, although only one HTTP method have been implemented. Requests with other method names will receive an HTTP error 405 (method not allowed). The message is self-descriptive as the index of any Web site is a starting point that lets the user know where to go from there. The service is stateless, it fetches all state from the home automation system behind the service, and the user of the service is oblivious to whether it is the home automation itself or an intermediary that returns the information.

#### 5.3.2.2 Device Class

By accessing one of the URIs in Listing 5.13 a device representation like the one in Listing 5.14 is returned. It is a JSON dictionary of the device properties, much like the dictionary the home automation system returns in the interface (see Listing 5.10). There is one difference though, instead of return the property's value, the value in this dictionary is a URI. This makes responses to a device cache-able, since it is unlikely that the device suddenly gains new kinds of properties while the system is running and it is also unlikely that a device's identifier (the `0` in the URI) will change during run-time, as the identifier should be maintained by the home automation system and not be subject to change by a user. Returning a cached response puts less strain on the home automation system and is a desirable property of RESTful Web services as mentioned in Section 3.4.1.

**Listing 5.14: An example of the JSON response to a GET request on '/(.*)'.**

```
{"on": "http://localhost:5050/0/on",
"identifier": "http://localhost:5050/0/identifier",
"name": "http://localhost:5050/0/name"}
```

The `Device` class also only allows `GET` requests, as can be seen in Listing 5.15. To update a device, `PUT` requests to the relevant attribute is issued. It is not REHAB's purpose to allow users to either add (`POST`) or remove (`DELETE`) devices from a home automation system, thus these methods are not implemented.

**Listing 5.15: The `Device` class in the exposure service.**

```
class Device:
    def GET(self, device):
        try:
            device = json.loads(interface.get_device_with_id(device))
        except Exception:
            raise web.notfound()

        properties = {}
        for property in device:
            uri = "{0}/{1}/{2}".format(
                    web.ctx.home,
                    device,
                    property)
            properties[property] = uri

        return json.dumps(properties)
```

The source code in Listing 5.15 is very similar to that of Listing 5.12, the only difference being that instead of returning a list of devices, a dictionary of the relevant device's properties is returned. The `GET` starts by trying to redefine the `device` argument (which is a device identifier) into a dictionary representation obtained through the interface. If the interface cannot return a device, an exception is thrown ("raised" in Python lingo) in which case the user is informed that device does not exist through an HTTP 404 error (not found).

#### 5.3.2.3 Property Class

The `Property` class responds to a URI like one of the ones shown in Listing 5.14 and returns the actual value of the property on a `GET` request. The `Property` class also implements a `PUT` method, since we want to enable update of devices' properties through the exposure service. The entire class is shown in Listing 5.16.

**Listing 5.16: The `Property` class in the exposure service.**

```
class Property:
    def GET(self, device, property):
        device = json.loads(interface.get_device_with_id(device))
        try:
            return json.dumps(device[property])
        except Exception:
            raise web.notfound()

    def PUT(self, device, property):
        interface.update_device(
```

```
11                  device_identifier = device,
12                  property = property,
13                  value = web.data())
```

The `PUT` method ought to utilize a try statement, as in the `GET` method, to allow for validation of the value input and return of a suitable error message to the user if the update fails. This is not the case due to time limitations. The `PropertyMeta` class referenced in Listing 5.11 simply returns the meta data that the interface returns, and its source code will not be listed here. The `Listener` class is equally simple, it can either return a list of current listeners via `GET` or register a new one with a simple call to the interface via `POST`.

## 5.4 Aggregation

Now that the simulation systems have been exposed, it is time to look at REHAB itself. REHAB functions as a hub for home automation systems that implement the REHAB interface and is exposed through the exposure Web service described in the last section. REHAB is fairly similar to the exposure service so instead of doing a thorough presentation of all the source code, I present the general structure of REHAB and go more into depth with a functionality that shows that changes in one home automation system may effect devices in another. The functionality is called "Rules".

### 5.4.1 General Structure

A general overview of REHAB and its communication with "REHAB enabled" home automation systems is depicted in Figure 5.5.

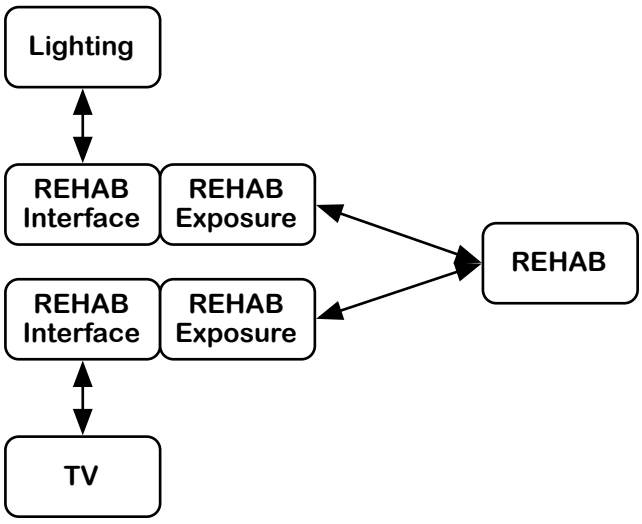


Figure 5.5: The communication to and from REHAB and home automation systems.

As showed in Figure 5.5, REHAB has taken the client role shown in Figure 5.4. The URIs REHAB responds to are show in Listing 5.4.2.

Listing 5.17: The URI scheme implemented in REHAB.

```
1 urls = (
2         '/', 'Index',
3         '/rules', 'Rule',
4         '/(.*)', 'System'
5         '/(.*)/(.*)', 'Device',
6         '/(.*)/(.*)/(.*)', 'Property',
7         )
```

All the classes, except `Rule` will be briefly described below, `Rule` is described in Section 5.4.2

**`Index`** Responds only to `GET` requests. Returns a JSON dictionary of the systems added to REHAB. The key in the dictionary is the system name (so no two systems can have the same name), and the value is a URI to the system's devices. The URIs are of the form defined in line 4 in Listing 5.4.2. Systems added to REHAB are serialized onto disk, so they will not be forgotten between launches of REHAB.

**`System`** Responds to `GET`, `POST`, and `DELETE` requests. `GET` returns a JSON dictionary, where the keys are URIs for the devices exposed by system specified (of the form specified in line 5 in Listing 5.4.2), the value is that device's name. HTTP `POST` requests adds a new system to REHAB, and `DELETE` requests removes the specified system. The `System` class ought to respond to `PUT` requests as well, but that has not been implemented.

**`Device`** Responds only to `GET` requests and return a dictionary similar to that of a `GET` request to a device in the exposure service.

**`Property`** Responds to `GET` and `PUT` requests and behaves identical to requests of those types to the exposure service.

### 5.4.2 Rules

REHAB has, as mentioned, a "Rules" feature, which shows that REHAB can be used to "mash up" home automation systems. Rules have the same form as an if statement, namely:

```
1 if rule's conditions are true:
2   execute rule's actions
```

The Rule module's model is displayed in Figure 5.6. The classes in the model is described in the following sections, beginning with `Condition`, ending with `Rule`.

#### 5.4.2.1 Condition

The `Condition` class has one method, which checks if the condition is true. It is shown in Listing 5.18.

Listing 5.18: The `check` method in the rule module's `Condition` class.

```
1 def check(self):
2     current_value = urllib2.urlopen(property_uri).read()
```

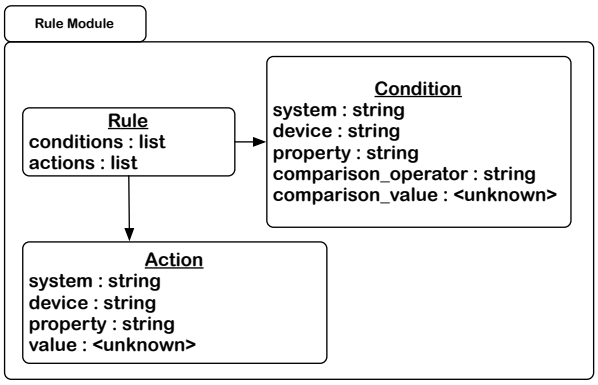


Figure 5.6: The Rule module's model.

```
3     current_value = json.loads(current_value)
4
5     if type(current_value) == int:
6         if not type(self._comparison_value) == int:
7             self._comparison_value = string.atoi(self._comparison_value)
8     else:
9         current_value = current_value.lower
10        comparison_value = comparison_value.lower
11
12    if self._comparison_operator == 'equals':
13        return current_value == self._comparison_value
14    elif self._comparison_operator == 'greater':
15        return current_value > self._comparison_value
16    elif self._comparison_operator == 'less':
17        return current_value < self._comparison_value
18    elif self._comparison_operator == 'not_equals':
19        return current_value != self._comparison_value
```

The `property_uri` variable used to fetch the current value of the device is built using the condition's `system`, `device`, and `property` variables. Once the value is fetched (and loaded via JSON) it is checked whether it is of type int, and if it is we convert the condition's `comparison_value` to an int as well. This is necessary, for if the comparison value were, for instance, "01", the current value were "1", and the operator "equals" the expression would not be true, even though that was the intention (Python compares string lexicographically using the numeric equivalents of each character). If the type is not int, the string is simply turned into lowercase and compared using the relevant operator. The method returns a boolean corresponding to the comparison statement.

#### 5.4.2.2 Action

Like `Condition`, the `Action` class has only one method method of interest, called `fire`. It updates the relevant property's value and is show in Listing 5.19.

**Listing 5.19: The `fire` method in the rule module's `Action` class.**

```
1 def fire(self):
2     property_uri = "{0}{1}/{2}".format(
3             system_uris()[self._system],
```

```
4             self._device,
5             self._property)
6
7     opener = urllib2.build_opener(urllib2.HTTPHandler)
8     request = urllib2.Request(property_uri, data = self._value)
9     request.get_method = lambda: 'PUT'
10    update = opener.open(request)
```

Here we see how to build the `property_uri` variable (the code omitted from Listing 5.18). It is done by fetching the system URI dictionary stored on disk by REHAB, and getting the relevant system URI from that dictionary. Then a request to that URI is built. The HTTP request issued has method `PUT` and the HTTP body contains the value that the relevant property should be updated to.

#### 5.4.2.3 Rule

The `Rule` class encapsulates a list of `Conditions` and `Actions`, and has two important methods called `check` and `fire`, they are both listed in Listing 5.20.

**Listing 5.20: The `check` and `fire` method in the rule module's `Rule` class.**

```
1 def check(self):
2     for condition in self.conditions:
3         if not condition.check():
4             return False
5     return True
6
7 def fire(self):
8     for action in self.actions:
9         action.fire()
```

They are very simple methods that simply utilizes the `check` and `fire` methods in the `Condition` and `Action` classes respectively.

The rule module is imported into REHAB so that every time REHAB is notified of a change in a home automation system, it will check all rules saved on disk, on the rules that are fulfilled will be "fired". This shows that REHAB indeed functions as a hub between home automation systems.

## 5.5 Client

The last piece of the REHAB puzzle is a client that enables "Mr. and Mrs. Smith" to benefit from the aggregation that REHAB performs. The client developed in this project is a Web application, but could just as well have been a desktop- or mobile application. The composition of all the systems is shown in Figure 5.7.

The client is an AJAX[11] Web application, meaning that is able to fetch data from a server asynchronously, without reloading the Web site. Asynchronous fetching of data is done using an API called `XMLHttpRequest` (XHR), which also enables the client to send HTTP `PUT` and `DELETE` messages, which are not included in HTML at this

[11] Used to be an acronym for Asynchronous JavaScript and XML. Now it is just a word, spelled "Ajax", that denotes a Web application with a desktop feel to it.

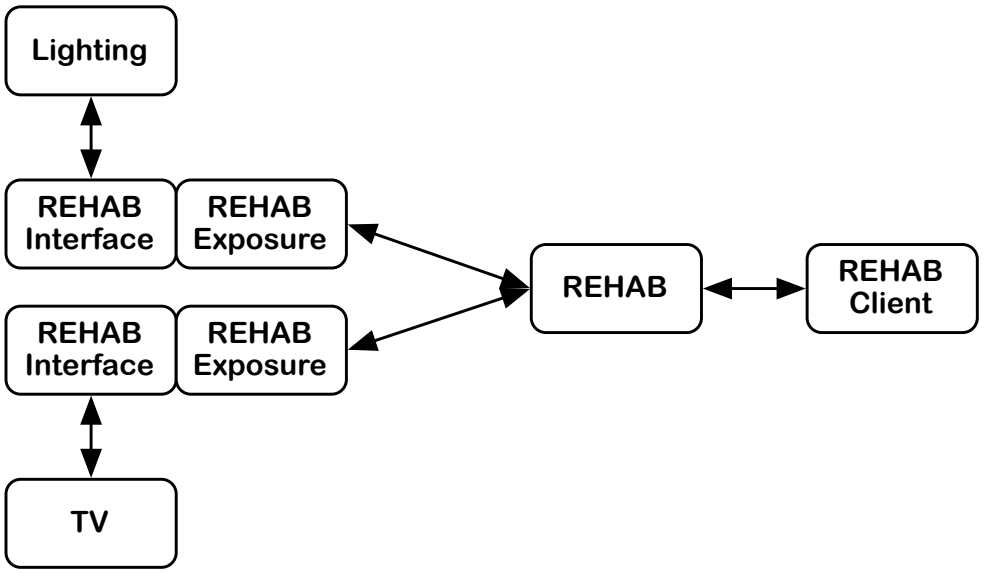


Figure 5.7: All the systems developed during this project put together.

point (they may be in HTML5[Hic10]). Before this project, I was not familiar with the principles involved in programming an Ajax Web application and it turned out to be a lot more complicated than I originally thought. I will not describe the entire client, it has more lines of code than any of the other systems developed during this project. Instead I present the general features (and layout), and the main Ajax issue that arose while building the client which has to do with application state.

### 5.5.1 General Features and Layout

The general layout of the client is shown in Figure 5.8. It consists of a site logo, a navigation area, a content area, and a loading indicator area.

The navigation bar is a list with three items: *Systems*, *Devices*, and *Rules*. The items are list themselves that function as sub-menus. When the user presses the *System* menu item, a sliding animation reveals the sub-menu consisting of an *Add* option, that enables the user to add a new systems to REHAB.

At the same time, when the user clicks the *Systems* menu item, the content area fades out, an asynchronous request for systems is made and the loading indicator, which has been invisible until this time, becomes visible with th purpose of informing the user that a request is being made. When the request finishes, the response is loaded into the content area, the content area fades back in, and the loading indicator is again made invisible. This is also the procedure when the user navigates to either *Devices* or *Rules*.

### 5.5.2 Application State Issue

When the user presses the *Systems* menu item, the application switches state from whatever page was being viewed previously to a state where it shows the systems in

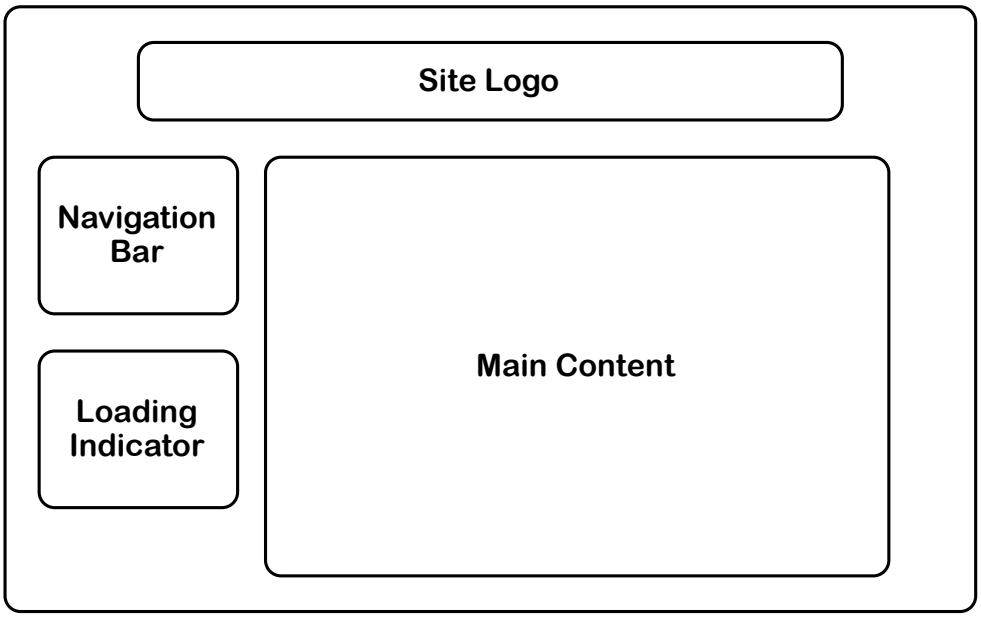


Figure 5.8: The layout of the client GUI.

REHAB. In synchronous Web applications the new state is reflected in the browser's address bar in form a new URI, for instance it changes from `http://localhost:8080/` to `http://localhost:8080/systems`.

When using the XHR object to load data asynchronously (instead of requesting an entire page and loading a new URI in the browser) data is fetched and loaded into the content area in Figure 5.8. Since the page is not *re*loaded, the window's location (the browser's address bar) is not being updated, which means that going back and forward using the browser's buttons for this does not yield the desired behavior. Instead of going back to the page previously being viewed, the browser redirects the user to the previous URI visited.

It is possible to update the address bar using JavaScript's `window.location` object to a new address, but that would cause the browser to reload that new address, which is what is trying to be avoided to give the application more of a desktop feel. What can be done instead is updating the `window.location.hash` which will not cause a reload. So for instance, let's say a user wants to link to the page with systems in the client, then the link could look like `http://localhost:8080/#/systems` instead of `http://localhost:8080/systems`.

However, when the user enters that link in the browser's address bar, the server serving the page does not receive the hash portion of the link, meaning it will serve the page corresponding to `http://localhost:8080/`. That page then has to implement a JavaScript function that can initialize the desired application state by animating the navigation to show the *Systems* sub-menu and display the loading indicator while making an asynchronous request to the server for the systems which are loaded into the content area when the request finishes.

This application state issue is one of many that I have encountered while programming the client. Chapter 8 goes into more detail about the experiences I have gained

from programming my first Ajax application.

# Chapter 6

# Test

There are two fundamental approaches to testing called, *black-* and *white box* testing [PI05]. White-box testing means that the tester has access to the source code, and the tester commonly write unit tests using this knowledge [Cop03].

Unit testing is the testing of the smallest unit of isolated code. A unit test in an object oriented environment commonly tests a single method by making assertions on what output is expected from the method given certain input. The purpose of unit tests is thus to make certain that a method behaves as expected even when given unexpected input, for instance a string instead of an integer or an integer above a certain threshold (for instance in the home automation system, a `Dimmer`'s `level` property should never exceed `100`). Generally all the "corners" should be tested. Unit tests are written for whole classes, meaning for every method in a class, and a so-called "driver" is written to automate the execution of unit tests.

This form of testing have not been employed in any of the systems implemented during this project. This is unfortunate as unit testing, from personal experience, is a good way to perform code review to both secure and optimize methods, but time simply has not permitted it.

The "black" in black box testing implies that the tester has no access to the source code, but can interact with the system being tested to see if it fulfills system requirements. Section 2.3.1 defines requirements for REHAB end-users (a.k.a. "Mr. and Mrs. Smith) and the system is tested in a black box manner, to show that it fulfills the functional requirements listed there, namely:

- Add home automation system to REHAB.
- Remove home automation system from REHAB.
- Display status of devices in said home automation systems.
- Update status of devices said home automation systems.
- Create a rule that shows REHAB is able to incur changes in one home automation system when a device in another home automation system changes status.
- Remove said rule.

To document that these functions is fulfilled by REHAB I resort to screenshots of the REHAB client in action. When entering the client URI in a web browser, the user is met with the screen in Figure 6.1.

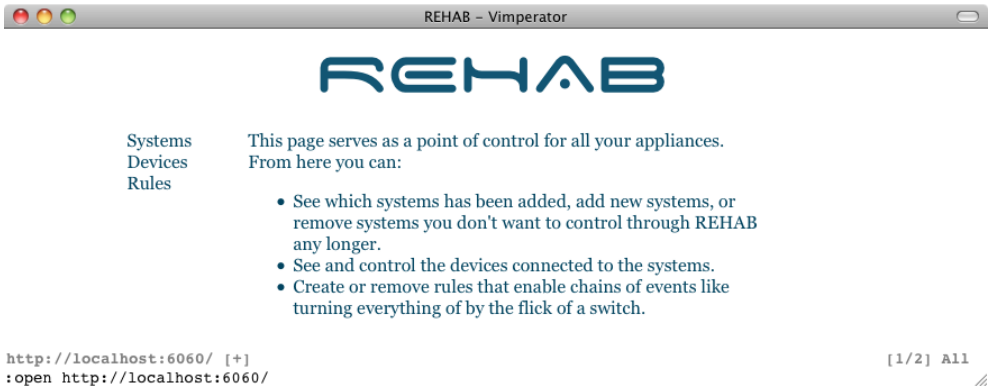


Figure 6.1: The start screen in the REHAB client.

## 6.1 Systems

This section presents the tests conducted regarding adding and removing a system, and thus its devices from REHAB. Figure 6.2 shows the screen the user sees when pressing the "Systems" link in the navigation bar, at this time there are no systems added to REHAB, and the user sees a text explaining this and how to add a system. Figure 6.3 shows the dialog in which the user can enter the details about a system to add. The details are the system's name and the URI of its exposure service, in this case the TV simulator is added whose exposure service is running on `http://localhost:3030/`. Figure 6.4 shows the screen presented to the user has added a system. At this time, the user may remove the system again by pressing the "Remove" button. If the user presses this button, another dialog is presented, shown in Figure 6.5, asking the user to confirm the removal of the system.

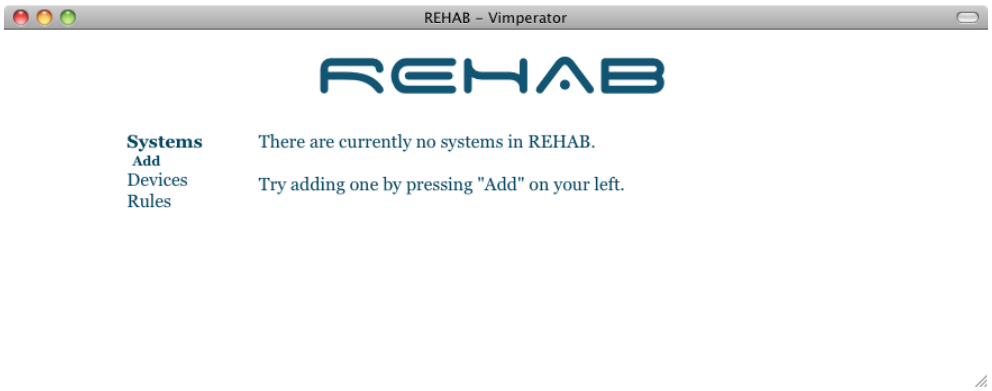


Figure 6.2: The system overview screen is initially empty, the user adds a system by pressing the "Add" link in the navigation bar.

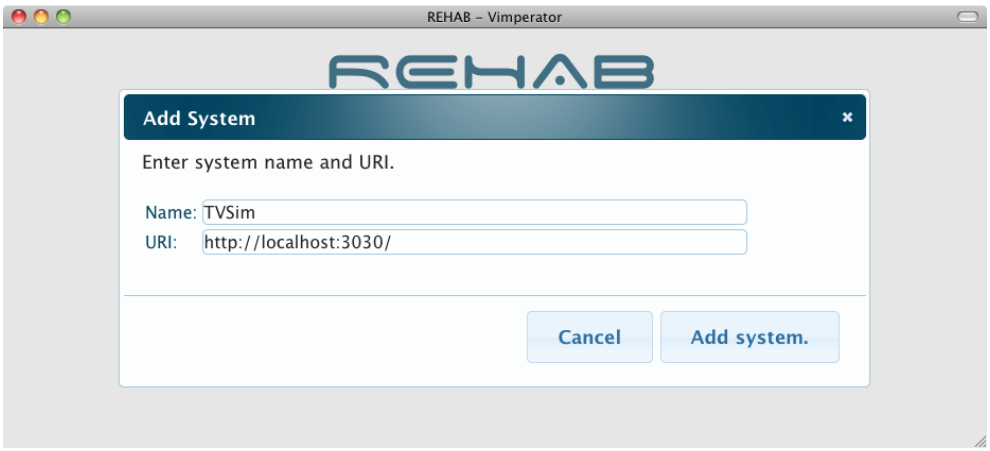


Figure 6.3: The dialog allowing input of system details.

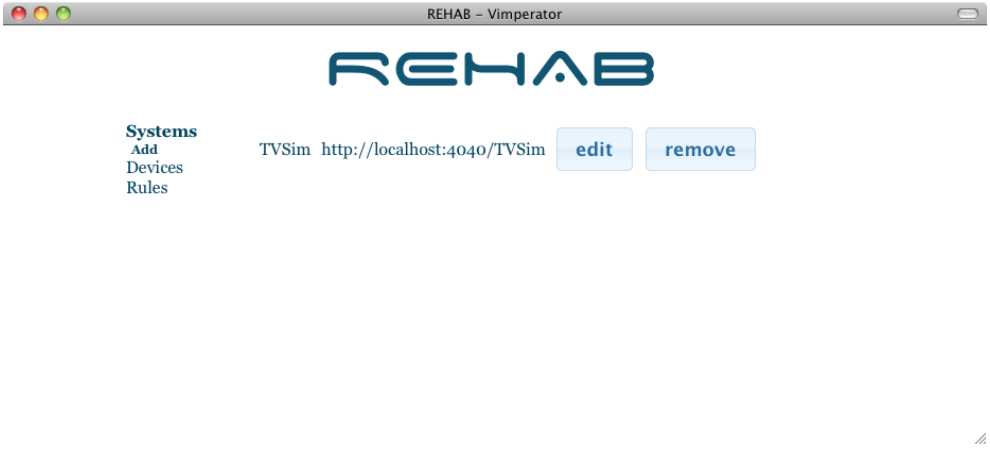


Figure 6.4: The updated system overview, the user can now remove the system again by pressing the “Remove” button

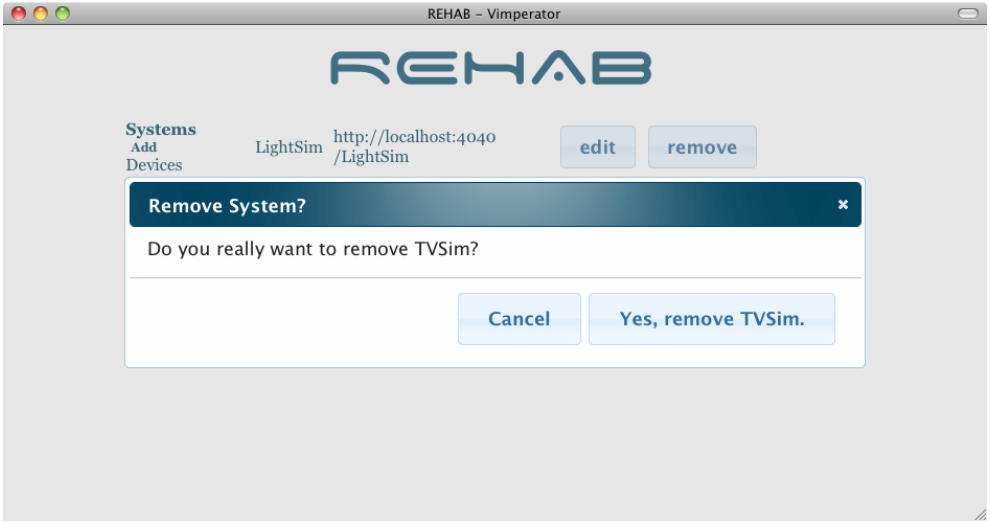


Figure 6.5: The dialog asking for system removal confirmation.

## 6.2 Devices

This section shows the tests that have been carried out to show that REHAB is able to present an overview over devices and their status and update that status as well. Before any of the following screenshots can appear, the user must first have added a system to REHAB, in this case the TV simulator has been added as shown in the previous section. Figure 6.6 shows a device overview. The device name is underlined by a horizontal rule and immediately following that ruler is its properties, in this case the name is "TV" and the properties are "volume", "name", and "channel". Figure 6.7 shows what has happened after the user has altered to volume property from 40 to 100 and pressed submit. Behind the scenes, the property has been updated, and the user is presented with the new current value.

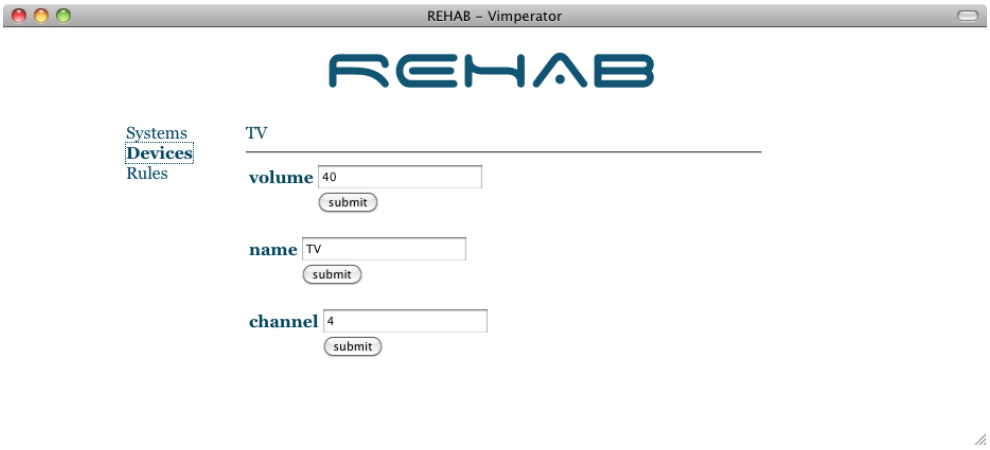


Figure 6.6: An overview of device and their status.

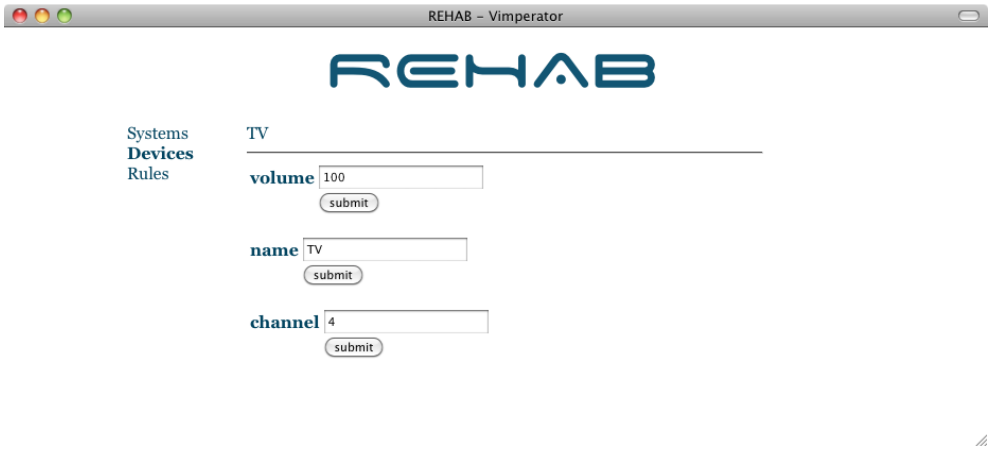


Figure 6.7: The overview after user has changed the volume property from 40 to 100.

## 6.3 Rules

The last end-user feature of REHAB is rules. The following screenshots assume that both the simulators developed during this project have been added to REHAB following the procedure described in Section 6.1. Figure 6.8 shows the screen when no rules have been created yet, it simply explains this fact and guides the user on how to create a rule. Figure 6.9 shows the dialog that allows the user to create a new rule. The user should input a name for the rule, and create at least one condition and action. The condition is created by choosing a device from the device list. Once chosen, the property list is updated to contain the relevant properties. After having selected a property, the user selects an "operator", in this case "is less than" and inputs a value, in this case "10". The action is created in much the same way. Figure 6.10 shows that if the user wants it, a rule can have multiple conditions and actions, they are added by pressing the plus buttons in the respective sections, the added conditions and/or actions can be removed again by pressing the minus button. After adding the rule, the overview is updated as is the case with systems in Figure 6.4.

After having created the rule that if the TV's volume is less than 10, the Switch should be turned off, we can test that the rule is actually put in effect by going to the devices overview. Figure 6.11 shows that the TV's volume is 90, and the Switch is currently on. Figure 6.12 shows what has happened after the user changed the TV's volume to 9, the Switch has turned off. Figure 6.13 shows the dialog in which the user is asked to confirm the removal of a rule.

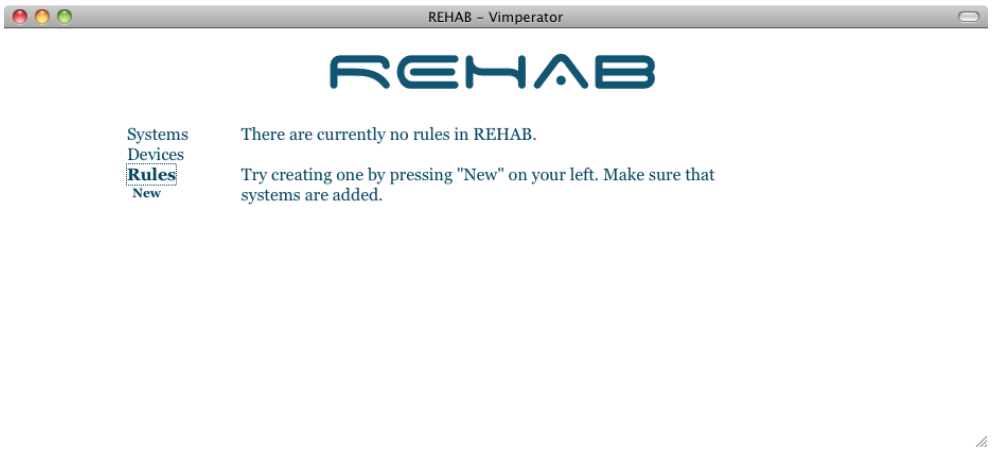


Figure 6.8: The initial rule overview, no rules have been created.

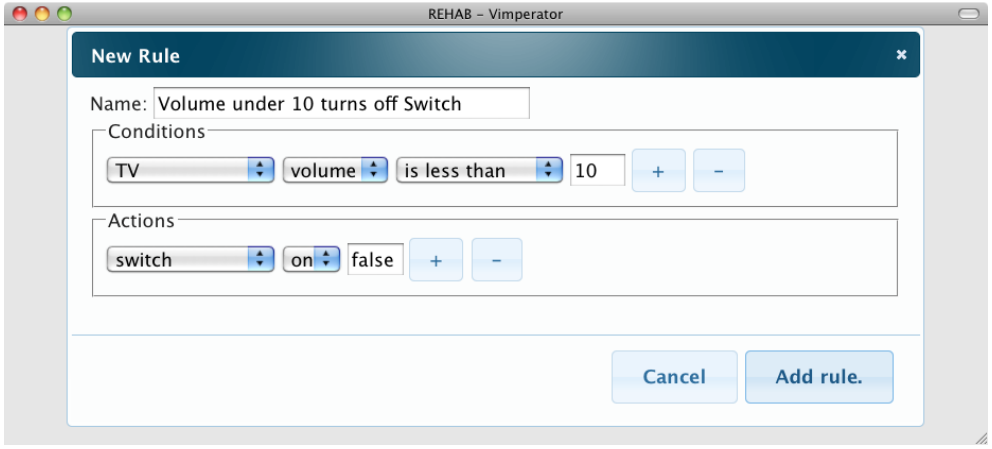


Figure 6.9: The dialog used to create new rules, in this case a rule with one condition, that the TV volume is less than 10, and one action, that the Switch should turn off.

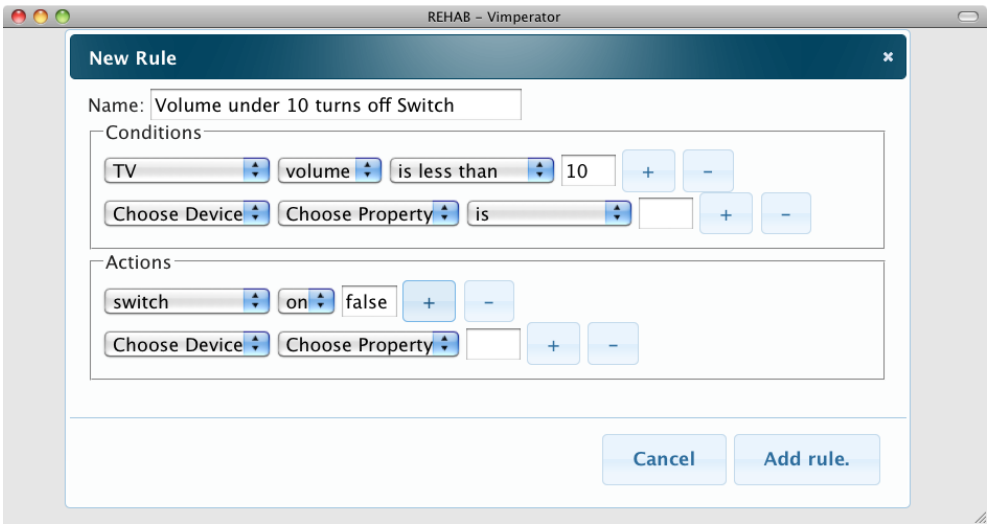


Figure 6.10: The user has pressed the plus button to add another condition and action.

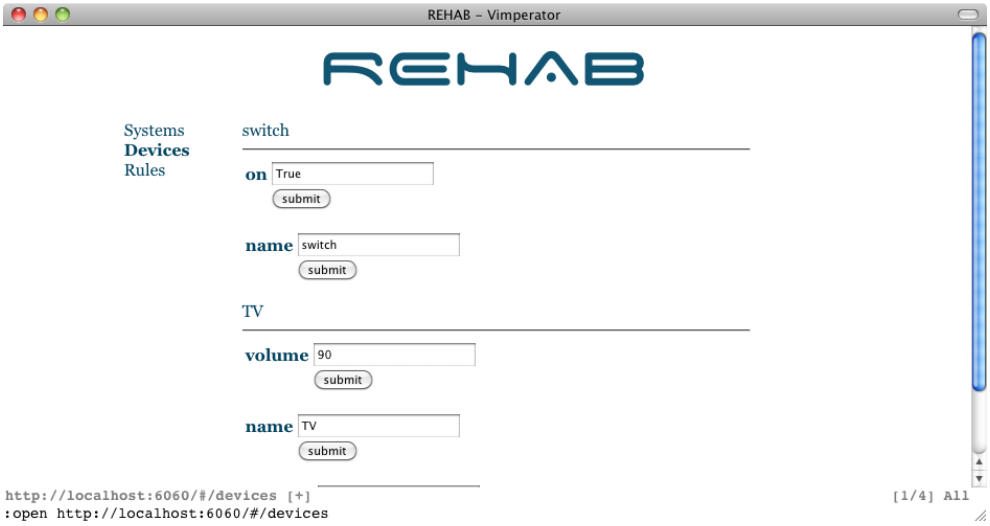


Figure 6.11: The devices before the rule has been executed.

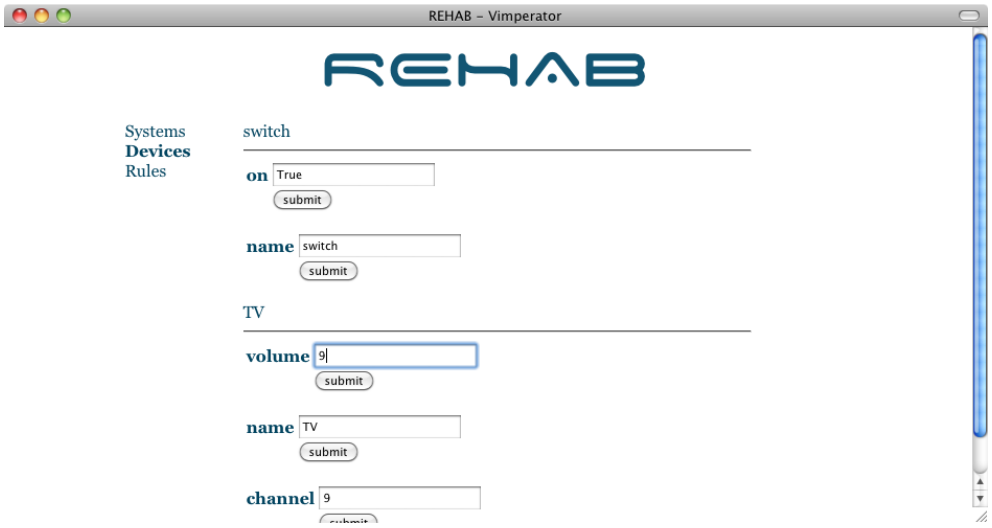


Figure 6.12: The devices after the rule has been executed.


Figure 6.13: The dialog box asking for rule removal confirmation.

# Chapter 7

# Conclusion

In this report I have documented my efforts to create a range of systems that enable integration of home automation systems in a platform based on already established standards. I have identified the requirements and implementation strategies for such a system in Chapter 2. I have reviewed the technical terminology pertaining to the identified implementation strategies in Chapter 3. In Chapter 4 I compare the strategies from a conceptual point of view, which enables me to make an informed choice between implementation strategies. In Chapter 5 I show the actual implementation of a number of systems that show one possible way of integrating home automation systems, and that the systems are implemented according the chosen implementation strategy. Chapter 6 documents testing of the finalized systems to show that they fulfill requirements identified in Chapter 2.

The hypothesis stated in Chapter 1 was that it is possible to create:

*A standardized communication platform, that is able to handle communication between many different kinds of home automation related systems.*

The main result is three systems that show the hypothesis to be true. The systems are my contributions with this project, they are:

**REHAB Interface** A simple interface that requires home automation developers to implement four methods in order to expose a home automation system to the Web. My findings is that it is sufficient to provide methods that return: a list of all devices, on device in particular, meta data about a device, and a method that facilitates updating a device. The interface further implements a "listener" mechanism, allowing third parties to be informed when a change in the system happens.

**REHAB Exposure** A system that exposes any home automation system that implements the REHAB Interface to the Web through a RESTful Web service. Home automation developers need only change a single line of code to enable this exposure. By implementing the service according to REST principles it exhibits a familiar interface, which is that of the Web. The familiar interface promotes easier adoption of a home automation system by third parties.

**REHAB Hub** A system that aggregates systems exposed with REHAB Exposure. REHAB Hub further exposes these systems, in a collected manner, through a RESTful Web service that enables central control of multiple home automation

systems through a client written for that purpose. REHAB Hub also implements a *Rule* feature that shows how changes in one home automation system can incur changes in another, whether the change happens through REHAB Hub or by means such as a remote control or wall switch within the home automation system.

To show that these three systems actually works, I have implemented two home automation simulators; one the simulate a home automated lighting system, and another that simulates a home automated TV set. The systems can be migrated to embedded platforms similar to those that host real world systems. Both systems implement the REHAB Interface, are exposed through REHAB Exposure, and aggregated in REHAB Hub. Lastly, I have implemented an Ajax Web application as a REHAB Hub client, that enable the user to add systems, monitor and change device status, and create rules.

During the implementation I have become familiar with C++, and become very fascinated with Python, two programming languages I had never touched before this project. I have learned how to implement a Web service that is loosely coupled from the system it exposes by following principles laid out by REST (which has guided the modern Web since 1994 [Fie00]). Thus, I have been standing on the shoulders on giants, which has given me valuable insight into the differences between two architectural styles, SOA and ROA. One detail that is particularly noteworthy is that REHAB has become a reality, not only through knowledge about these architectural styles, but also by implementing the link between them using object oriented principles. The REHAB Interface is comparable to interfaces, also known as protocols, in Java or C# for instance. The listener mechanism in REHAB Interface employs the same principle as observer pattern described in

With regard to the project goals stated in Section 1.4 they have all been accomplished as described in the introduction to this chapter. The main goal: “to learn new things”, has also been accomplished and is demonstrated by the amount of programming languages and technologies I were unfamiliar with before this project.

# Chapter 8

# Future Work and Evaluation

This chapter evaluates the work that this report documents, especially the lessons learned from the technologies used during implementation, and suggests future work to the systems developed during this project.

## 8.1 TV Simulator

The TV simulator, written in C++, unfortunately lacks the ability to receive changes through native controls, which is simulated by a socket running in a separate thread in the lighting simulator. I have spent countless hours trying to implement a more sophisticated TV simulator that had this ability, but ended up with a simple system due to time limitations. To get threads and sockets working cross-platform requires a great deal of work due to operating system specific implementation of these features. I tried to use Boost library (I already use the Boost.Python library to compile a Python module based on C++ code), but simply used too much time getting it to work. I have to admit that I find that C++ has a very steep learning curve, mainly due to its syntax, pointers, and memory allocation/deallocation. I did not expect this to be as big a challenge as it turned out to be because of my knowledge about Objective-C, which I have used for iPhone application development in an earlier project, but I was mistaken; it was a much larger challenge than anticipated.

The Boost.Python library also took some time to understand, but was not nearly as complicated as I had expected. It introduces no new syntax to C++, which is the case with similar libraries. It also comes with comprehensible documentation and examples that ease adoption greatly.

## 8.2 Lighting Simulator

The lighting simulator implements the REHAB interface fully, and is able to receive updates via native controls, simulated through a socket running in a separate thread. The simulator exhibits the same features as a home automation system that I am already familiar with, called Innovus.

The simulator is implemented in Python, which is also used to implement a real world home automation system project at Aalborg University, called HomePort. HomePort uses C programs to send messages in a wireless fashion using the ZigBee protocol. As such, I find the lighting simulator to be an accurate facsimile of a real world system.

The lighting simulator was my initial introduction to Python. I have previously worked with dynamically typed programming languages, which has made me firm believer in statically typed programming languages. Too often, I have found myself in a situation expecting one result, gotten another, and consequently wasted hours of perfectly good time. Thus, I was apprehensive about using Python, but have not experienced any significant problems using Python. I found it to be an easy language to learn with many libraries that seem to be well-documented.

## 8.3 REHAB Interface

The main concern with REHAB Interface was to determine which functionality it should provide, how it could be implemented in the simulation systems, and how it could be used to expose those systems. Realizing that the interface was not that different from interfaces used in object oriented environments, eased to development process greatly as it was then a matter of applying patterns from the object oriented world.

## 8.4 REHAB Exposure

It would have been nice to have time to implement a standard client that could enable interaction with home automation systems exposed through REHAB Exposure, but time has not permitted it. When exposing home automation systems to the Web, a lot of security concerns is also introduced. To accommodate these security concerns, the exposure service should implement user verification and communicate over a secure connection, e.g. by using the HTTPS protocol that combines HTTP and SSL.

As it is now, a REHAB Hub user needs to know the base URI of the exposure services running. It is very conceivable that "Mr. and Mrs. Smith" is unfamiliar with the URIs serving a home automation system. Thus it would be nice to implement a service discovery protocol like the IETF authored "Zeroconf"[1]. Apple has their own implementation of zeroconf, called Bonjour, which locate e.g. printing services on a local network.

## 8.5 REHAB Hub

The "Rules" feature of REHAB Hub requires more attention than there has been time for in this project. One concern that needs to be addressed is the possibility for loops in rules. This can be explained by a very simple example, imagine having the following two rules:

[1] http://zeroconf.org

1. If switch is on, turn it off, and
2. If switch is off, turn it on.

Such rules contradict each other and causes the lamp connected to the switch to be turned on and off indefinitely. To accommodate this problem a procedure checking for loops in rules can be implemented by doing the following when a new rule is being created:

- Make sure to have a list of all existing rules.
- Fire the new rule's actions.
- Check to see if any rule in the list of existing rules is fulfilled.
- If there is a fulfilled rule, execute its actions and put the rule in a list of executed rules.
- Now check the list of executed rules, to see if any rule in that list is fulfilled.
- If no rule in the executed list if fulfilled, continue the check in the list of existing rules.
- If a rule in the executed list is fulfilled at any time, the new rule causes a loop and should not created.

Another issue concerning rules is that devices may not behave is the user expects. Imagine the following rule: (from Chapter 6)

- If volume is less than 10, turn off switch.

If the user wants to turn off the lamp when the volume is less than 10, the lamp will immediately be turned off again because rules are checked. This can be avoided by, while checking rules, making sure that a rule to be fired has no effect on the just updated device, and if it has not firing the rule. However, that approach would only be a temporary solution since the next device update to occur would also trigger a checking of rules and cause the switched to be turned off. All in all, a more elaborate rule system is required.

## 8.6 Ajax Client

The Ajax client that enables a user to control systems added to the REHAB Hub. As stated earlier, it has more lines of code than any other system implemented during this project and it has a lot of quirks. This is a result of me not being very familiar with JavaScript before this project, and it turned out to be a bigger challenge than expected. One thing that would improve the user experience is more robust controls to update a device's status. Currently, the user has a text field to his disposal, and whatever is entered into the text field is transmitted to the home automation system relevant to the device. If the home automation system lacks safeguards against accidental input this may cause errors in the home automation system. Figure 8.1 shows more user friendly controls to update device states. The controls are loaded based on the meta data provided through REHAB Interface, but unfortunately time ran out and they are not functional, i.e. does not initialize to to correct values, nor do they have any impact on device states.

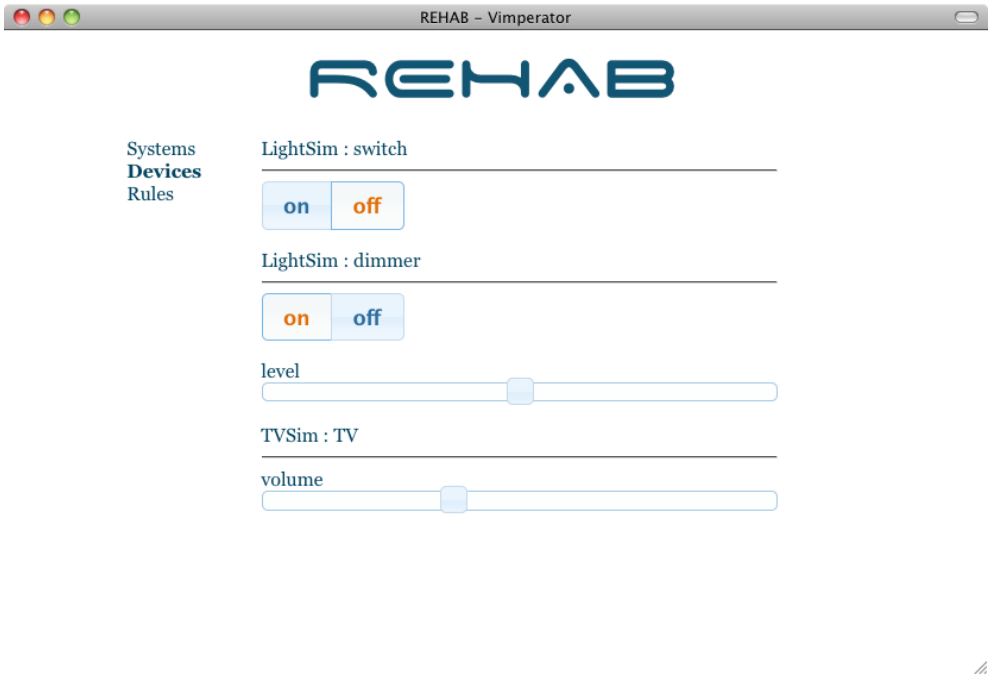


Figure 8.1: More user friendly GUI controls to control device states.

# Bibliography


[AA10] S. Allamaraju and M. Amundsen. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. Oreilly & Associates Inc, 2010.

[AKL+06] Nikolaj Andersen, Thomas Legaard Kjeldsen, Christiand Planck Larsen, Morten Vejen Nielsen, and JÃ¸rn Martin Rasmussen. A Technical View on SOA and Related Acronyms. Online: `https://services.cs.aau.dk/cs/tools/library/details.php?id=1166605744` [Accessed February 2010], 2006.

[BP+] T. Bray, J. Paoli, et al. Extensible Markup Language (XML) 1.0 (5th edn.). W3C Recommendation (November 26, 2008). Online: `http://www.w3.org/TR/2008/REC-xml-20081126/` [Accessed February 2010].

[CHvRR04] L. Clement, A. Hately, C. von Riegen, and T. Rogers. UDDI Version 3.0. 2 (UDDI Spec Technical Committee Draft). 2004.

[CL02] E. Cerami and S.S. Laurent. *Web services essentials*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.

[CM08] Yanik K. Challand and Tim M. Madsen. Context Aware Device Control. Online: `https://services.cs.aau.dk/cs/tools/library/details.php?id=1229376701` [Accessed February 2010], 2008.

[CM10] Yanik K. Challand and Tim M. Madsen. Using Rule Engines in Home Automation. Online: `https://services.cs.aau.dk/cs/tools/library/details.php?id=1262858090` [Accessed February 2010], 2010.

[CMRW07] R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Online: `http://www.w3.org/TR/2007/REC-wsdl20-20070626/` [Accessed February 2010], 2007.

[Com] Python Community. Python/c api reference manual. Online: `http://docs.python.org/c-api/` [Accessed May 2010].

[Con00] UDDI Consortium. UDDI Executive White Paper. Online: `www.uddi.org/pubs/UDDI_Executive_White_Paper.pdf` [Accessed February 2010], 2000.

[Cop03] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Incorporated, MA, USA, 2003.

[DZ83] JD Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[Erl05] T. Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext Transfer Protocol–HTTP/1.1. *RFC Editor United States*, 1999.

[Fie00] R.T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, Citeseer, 2000.

[For82] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem* 1. *Artificial intelligence*, 19(1):17–37, 1982.

[GEI06] MT Galeev, S. Engineer, and M. Inc. Catching the Z-Wave. *Embedded Systems Design*, 19(10):28, 2006.

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, H.F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP version 1.2 part 1: Messaging framework. Online: `http://www.w3.org/TR/soap12-part1/` [Accessed February 2010], 2007.

[GKC04] N. Gershenfeld, R. Krikorian, and D. Cohen. The Internet of Things. *Scientific American*, 291(4):76–81, 2004.

[Hic10] Ian Hickson. HTML5, A vocabulary and associated APIs for HTML and XHTML. Online: `http://dev.w3.org/html5/spec/Overview.html` [Accessed June 2010], 2010.

[Hoa] L.N. Hoang. Middlewares for Home Monitoring and Control.

[Kay03] D. Kaye. *Loosely coupled: the missing pieces of Web services*. RDS Strategies LLC, 2003.

[Kin03] P. Kinney. Zigbee technology: Wireless control that simply works. In *Communications Design Conference*, volume 2, 2003.

[Kle01] J. Klensin. RFC2821: Simple mail transfer protocol. *RFC Editor United States*, 2001.

[Luo98] A. Luotonen. Tunneling TCP based protocols through Web proxy servers. *Work in Progress*, 1998.

[Mad09] Tim M. Madsen. Distributed and Mobile Application Development. Online: `https://services.cs.aau.dk/cs/tools/library/details.php?id=1243439802` [Accessed February 2010], 2009.

[ML03] N. Mitra and Y. Lafon. Soap version 1.2 part 0: Primer. *W3C Recommendation*, 24, 2003.

[MMMNS00] L. Mathiassen, A. Munk-Madsen, P.A. Nielsen, and J. Stage. *Object-Oriented Analysis and Design*. Marko, Aalborg, 2000.

[NKMHB06] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web services security: Soap message security 1.1 (ws-security 2004). *OASIS Standard Specification*, 1, 2006.

[Par72] DL Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1058, 1972.

[PI05] R.S. Pressman and D. Ince. *Software engineering: a practitioner's approach*. McGraw-Hill New York, 2005.

[PW09] C. Pautasso and E. Wilde. Why is the web loosely coupled?: a multi-faceted metric for service design. In *Proceedings of the 18th international conference on World wide web*, pages 911–920. ACM, 2009.

[PZL08] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[RGK90] J. Radatz, A. Geraci, and F. Katki. IEEE standard glossary of software engineering terminology. *The Institute of Electrical and Electronics Ehgineers, NY, USA*, 1990.

[RR07] L. Richardson and S. Ruby. RESTful web services. 2007.

[Str] B. Stroustrup. An overview of the C++ programming language. *Handbook of object technology*.

[W3C04] Working Group W3C. Web Services Glossary. Online: `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/` [Accessed February 2010], 2004.

[Wie03] K.E. Wiegers. *Software requirements*. Microsoft Press Redmond, WA, USA, 2003.