JORRIT DE BOER

SMART OFFICES: A WEB-BASED INTERFACE FOR TASK SELECTION

[July 14, 2013 at 13:41 – classicthesis version 1.0]

SMART OFFICES: A WEB-BASED INTERFACE FOR TASK SELECTION

JORRIT DE BOER

In Partial Fulfilment of the Requirements for the Degree of Bachelor in Computing Science

July 2013 – version 1.0

SUPERVISORS:
Daily supervisor: Ir. I. Georgievski
First supervisor: Prof. Dr. Ir. Marco Aiello
Second supervisor: Prof. Dr. Ir. Paris Avgeriou

Jorrit de Boer: *Smart offices: A web-based interface for task selection,* In Partial Fulfilment of the Requirements for the Degree of Bachelor in Computing Science , © July 2013

ABSTRACT

This thesis presents the approach taken to develop a web application for a smart office. The proposed interface attempts to make the role of the smart office user more active by creating a user interface which is user-friendly: unintrusive and easy to use. The interface proposes to enable selecting complex tasks by enabling easy and natural interaction.

Several extensions are proposed for further research such as using representative images for tasks, sustaining the hierarchy of the environment, using images for environmental objects, connecting the interface to physical devices and adding voice input features.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

MVC Model-View-Controller

PDDL Planning Domain Definition Language

HPDL Hierarchical Planning Domain Language

JSON JavaScript Object Notation

HTML HyperText Markup Language

CSS Cascading Style Sheet

SHP Scalable Hierarchical Planner

REST REpresentational State Transfer

1

INTRODUCTION

In a high-tech world where technology is integrated into our society, information gets processed automatically without the collusion of the user. Technologies have evolved rapidly and the user is confronted with becoming familiar with the technologies used. Because offices are used for multiple purposes (e. g., conducting research, holding meetings) and electronic devices are the rule rather than the exception, it is worth investigating to what extent office work can be automized.

A smart and energy-efficient office makes decisions based on the environment and other knowledge it possesses. The decisions autonomously being made by the smart office for the sake of the office occupant do not always lead to desired behaviour, since the user might think the decisions being made by the smart office do not support and even disturb the working process.

At the moment the smart office user does not have enough influence on in what way the smart office should respond [2]. A solution to giving the user the opportunity to specify what should happen in the office is to let the user define tasks which represent what the user can accomplish in the office. Users should be able to choose a combination of simple and more complex tasks (compound tasks). The last category consists of a sequence of simple and/or complex tasks. The selection of tasks can be enabled by creating a web-based interface which should be intuitive and thus easy to use for non-experts. Enabling user input to the smart office in a very intuitive way makes sure the application as a whole is appealing to a broad public. Since the interface is intended to be implemented using a web application, the interface will work regardless of the operating system used and the hardware present. Furthermore, the end user is most likely familiar with using a computer and browser which is advantageous compared to difficult to handle devices and software.

Interface design on its own is an interesting topic to study, since there is a lot of freedom in designing an interface and the quality of the interface will stand or fall with the user's judgement. One of the challenges the development of a graphical interface is facing is building the interface in such a way that it has enough functionalities and

is user-friendly at the same time. Adding too many functionalities will lead to an application which is too complex. As a result, the user feels overwhelmed by information. On the other hand, providing not all the desired functionalities will not satisfy the user either.

The aim of this thesis is to develop a web-based application which assists the user in selecting tasks, but which also has other relevant functionalities. Examples of relevant functionalities are displaying the current state of the environment and showing the user's preferences. This thesis investigates the possibility of developing a web application for smart offices which has the following objectives:

- Make the role of the smart office user more active: the smart office autonomously makes decisions which do not always satisfy the user's needs
- Create a user-friendly interface which is unintrusive and easy to use
- Provide the ability to select complex tasks and to show the environmental state
- Enable easy and natural interaction such as speech-based interaction

1.1 THESIS STRUCTURE

The methodology used is described as well as a justification of the decisions made during the project. At first some theory should be described in order to understand the context of the problem. Chapter 2 describes smart environments in general, defines a smart office and describes related projects. The design of the user interface as well as important concepts are described in Chapter 3. The concepts also involve a description of existing systems to integrate with. This chapter contains a specification of the requirements and architecture to be used as well. The technical details are elaborated upon in Chapter 4, in which all the technologies used are described. In Chapter 5 the conclusions of this thesis are stated and some suggestions for future work are introduced.

2

SMART ENVIRONMENTS

In a smart environment smart devices are cooperating for the sake of the office occupants. Smart refers to the ability to autonomously acquire and apply knowledge, while environment is the notion of our surroundings [3]. There are many kinds of smart environments, examples include, but are not limited to, smart homes and smart rooms. In the rest of this Chapter smart offices are discussed first followed by a description of related projects.

2.1 SMART OFFICES

A smart and energy-efficient office makes decisions based on the environment and other knowledge it possesses. An example of this knowledge is which (electrical) appliances are available in the office and which actions are permissible. The main goal of a smart office is to support the office users in an unintrusive manner. There are many types of support such as automizing daily office tasks, ensuring the user's safety and maximizing daily profit. The user needs to decide whether it finds the actions performed by the smart office supportive or not.

According to Cook et al. a smart office is not just another smart environment: it is a very specific type of smart environment which distinguishes itself from other types of smart environments on several characteristics:

- "Office work is usually highly automizable and computer-related"
- Offices can be used for many purposes (e. g., meeting rooms, discussion rooms etc.)
- Office users have different levels of (computer) skills (e. g., a simple end-user is satisfied with a simple menu, but a more advanced user desires a more advanced menu)
- The interaction with smart offices depends on the type of user (e. g., a user might prefer speech- based interaction)

2.2 RELATED PROJECTS

Although the author is unaware of related projects involving development of a web-based interface, existing related projects on the field of smart environments were already considered. Even though some of the related projects are based on smart homes, the ideas introduced in this type of project can be used as inspiration for designing smart office interfaces as well.

2.2.1 *Interface Design*

In order to assist office users in doing their everyday tasks it is important to design a user-friendly user- interface in which the user has the possibility to control the environment. Several related projects on the field of interface design for smart environments are described in the following subsections.

2.2.1.1 *MASP*

The Multi-Access Service Platform [20] allows "users to evaluate and control multimodal interaction". This system has as its main goal to give users a better logical understanding of complex smart environments. The system also supports multimodal interaction, so not only the 'usual' input methods (e. g., keyboard and mouse input) but also speech input can be used. The interface has a lot of configuration settings which can be adjusted by the user. Therefore, the user is able to adapt the system to its needs. This advantage overcomes the problem that users do not have enough influence on the actions performed in a smart environment. A screenshot of the system is provided in Figure 1. The system can automatically detect changes in the availability of resources, thus services can be added or removed as time proceeds. Even though the MASP is not a web-based application, ideas from it can be used for the development of the smart office interface described in this thesis.

2.2.1.2 *Empirical Evaluation of Existing User Interfaces*

According to Koskela et al. the added value of their research on existing user interfaces is that several user interfaces are compared by testing the interfaces in an actual living environment over a long period of time. Previous research did not take into account realistic test

Figure 1: Screenshot of the MASP user interface [20]

scenarios and often the testers of the interface also belonged to the interface design team.

Although Koskela et al. distinguish three phases in their paper, in this thesis only the evaluation phase is discussed, because it is useful information for designing a smart office interface.

The results of the research put forward the need for two different types of user interface requirements for tasks. *Pattern control* enables the timing of events, e. g., prepare the home (or office) for daily routines. The other type of requirement for tasks is *instant control*: impulsive tasks which should be executed immediately. Whenever the user wants to control a device, the device should be ready to handle the user's request. Furthermore, the user should be able to control frequently used devices by taking only a couple of steps (e. g., by using shortcuts). It is also important that the user can control the environment on a single place, e. g., closing the curtains, turning on the projector and dimming the lights without having to pass through the home or office.

The smart home research project showed that the test subjects (a couple) particularly liked the user interface of the mobile phone (displayed in Figure 2), especially because it allowed them to control the environment without having to move to a smart home control device (e. g., a computer containing smart home control software). Although this has not been tested, it is higly probable that this problem does

Figure 2: Example of a mobile phone user interface [15]

not appear in smart offices, because a smart office is relatively small compared to a smart home, so it is relatively easy for the user to move to a smart office control device (e. g., the web-based application installed on a desktop computer).

An important lesson drawn from Koskela et al.'s paper is that user's expectations are not always met and that unforeseen problems arise. In summary, the empirical evaluation of existing user interfaces brings to light several interesting conclusions which can be used as inspiration for designing a smart office interface.

2.2.2 *Smart Office Applications*

In the paper of Ramos et al. several existing smart office applications are described. In this subsection, the most important and relevant projects described in that paper is elaborated upon.

2.2.2.1 *Monica SmartOffice*

The Monica SmartOffice [16] intends to "anticipate user intention" and "communicate useful information" by using "user monitoring". The project involves placing many microphones and cameras in the office. The cameras were mainly used for face recognition and user activity tracking. The microphones serve for capturing user speech input. The project also involves software modules, each having a specific task.

2.2.2.2 *Intelligent Environment Laboratory of IGD Rostock*

Whilst most systems use a function-oriented interaction, this laboratory uses a goal-oriented interaction. The intention of the laboratory is "to create an interactive environment based on multimodal interfaces". Each component provides "a description about the meaning

and effect of their function". The semantic descriptions are implemented as "a set of preconditions that must be fulfilled" in order to execute an action. The semanting description of the effects is passed to "a planning assistant" which calculates the plan needed to fulfill the goals. The Intelligent Environment Laboratory of IGD Rostock stands out because goals should be reached instead of actions should be performed.

2.2.2.3 *Smart Doorplate*

The Smart Doorplate project "is able to display notes for visitors and change them remotely". For example, this project can be used by an office occupant to leave important messages whenever he/she is not present. Several devices, such as displays, sensors, microphones and person tracking systems, are present in the office to support the goal of the Smart Doorplate project: provide the office visitor with useful information in case of owner absence.

3

CONCEPTS AND DESIGN

It is important to have a good understanding of the concepts necessary to develop a working interface for smart offices. The design of the interface is a crucially important subject which requires thoughtful decisions. The application as is needs to be integrated with many existing components. These components are explained together with requirements developed taking into account the existing components. Based on the requirements, an architectural concept is created. The design decisions are justified using supporting screenshots of the application.

3.1 AN OVERVIEW

Figure 3 provides an overview of the role of the smart office interface with respect to existing components.

As is described in more detail in section 3.2, the *planner* needs domain and problem specifications. The *domain* component is needed by the planner in order to provide knowledge about how to solve a decision-making problem. Basically, the problem specification consists of two parts: initial state information and task specification. The initial state information is stored in the repository component. This component sends the information to the planner via the *Smart Office Interface*. The *repository* component represents a database which contains information about available objects. The user of the smart office interface selects tasks and the task specification is generated by the interface based on the selected tasks. Whenever the user wants the selected tasks to be executed the planner is invoked, it gathers the domain and problem specification and calculates the actions taken as a result of execution. In case the state of the environment changes, an update is sent by the *Environment State Change Notifier* and the repository gets updated.

Each component of this scheme is explained in more detail in the following sections.

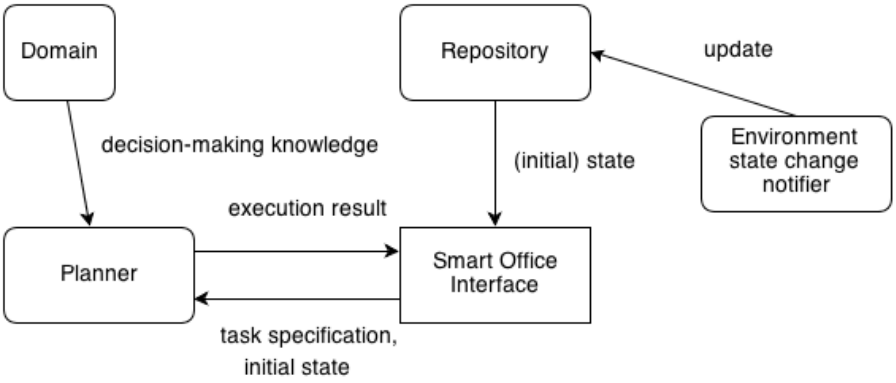

Figure 3: An overview of the existing components

3.2 PLANNING

A smart environment needs to make a lot of decisions which anticipate the users needs. Even though an office is often relatively small, the number of actions possible is still large. There are often many ways to achieve the same goal and it is not always clear which way is the best. User-defined goals are often complex and the smart office needs to find a way to fulfil them. It is important that the user should not be limited in the selection and execution of goals (e. g., not only predefined goals should be possible). Furthermore, the application needs to be able to deal with different smart offices which contain multiple devices which can be arranged differently. In summary, a smart office autonomously needs to decide which steps to take for the sake of the office occupant, so a decision-making system is needed.

3.2.1 *HTN Planning*

One way to support the decision-making process is to use a system based on HTN planning. HTN planning is a planning methodology used in the field of Artificial Intelligence in which the planning system repeatedly decomposes subtasks until only primitive tasks (actions) are left [8]. The key components of a HTN planning system are tasks, methods and actions [22]. These essential terms are explained in detail in section 3.2.1.1.

Using HTN-planning has a lot of advantages [12] and a system using this approach is "particularly well-suited for adaptable and user-centric environments" [14]. A smart office conforms to these characteristics, so this type of system is suitable for the application. The system

is referred to as the *planner*, because one system component relies on the use of planning techniques from the AI field. The planner is in fact the implementation of (HTN) planning and this is elaborated upon in section 4.4. Figure 4 shows a simplified representation of the planner by regarding it as a black box. According to Kaldeli et al. the planner makes sure that the user focuses only on declaring the desired tasks, thus the user does not have to care about how the tasks are achieved.

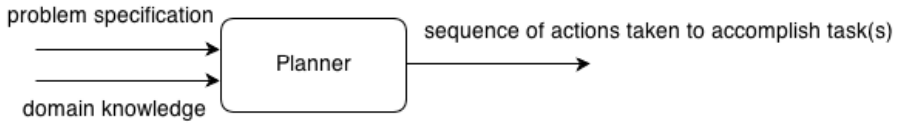

Figure 4: A very simple black box representation of the planner

3.2.1.1 *Types of Tasks*

Smart office users want to select different types of tasks which mainly differ in complexity. Many tasks require involvement of multiple devices and are decomposed into simple and/or complex tasks. An action is defined as a simple task. Actions represent the building blocks for tasks. They can be seen as a simple task with an observable effect. As an example, consider switching a light. This action has an observable effect since the user can see the light being turned on or off.

definition of an action

A smart office only providing the ability to select actions is not sufficient for a smart office user because in that case the smart office does not have any added value: the user can select and execute actions easily without involvement of the interface. The added value of the interface is that it provides a way for users to select and execute complex tasks. Therefore, users should be able to select more complex tasks as well. A complex task is defined by multiple methods. A method is conveniently defined as a possible way to achieve a (complex) task. A method is built out of simple or complex tasks.

definition of a complex task

definition of a method

The concepts introduced can be best illustrated using an example. Suppose the smart office user wants to hold a meeting in his/her office. The office needs to be changed to a meeting room, so the user selects the task *set-meeting-room*. Setting the meeting room can be done in multiple ways since it depends on the current state of the environment. For example, one method of the task represents the actions to be executed in the case the level of (natural) light emission is too high. A different method could be handling the case in which the level of light emission is adequate. The last mentioned method dims

the lights and activates the projector. The first mentioned method also activates the sun shade in order to compensate for the high level of light emission.

3.2.2 *Domain*

The planner needs domain knowledge in order to make decisions. Domain knowledge can be described as knowledge about the current environment like which objects are present in the office and what are the allowable actions. Consider the following example for a clarification of domain knowledge.

Suppose the user wants to execute the task *type-a-document*. The planner first selects the *type-a-document* task and checks if this task has a method satisfying the precondition (a condition which needs to be satisfied in the current environment before decomposing the method). If there is an applicable method, the method is applied by decomposing it into one or more tasks. *type-a-document* is a complex task which has only one method which is decomposed into the actions *turn-on-computer*, *turn-on-desk-light*, *log-user-in* and *start-word-processor*, because the precondition (e. g., the computer is in the user's office) is satisfied. The fact that the task *type-a- document* is subdivided into the just mentioned actions is an example of domain knowledge.

3.2.3 *Problem*

Providing the planner only domain knowledge does not make sense, since the planner needs to know for which problem it needs to make decisions. Basically the problem consists of two parts: information about the initial state and a specification of one or more tasks. The initial state represents the state of the environment just before the task is selected. Initial state information includes the following information:

1. Available locations and their types (e. g., building32 is a building, room22 is an office)
2. Available objects and their types (e. g., light34 is of type light, proj05 is of type projector)
3. Current value of the object (e. g., 70%, turned-on/turned-off)

4. Location of the object (e. g., light34 is in room534, office368 is on floor02)

The specification of the task(s) is the main focus of this thesis and should be enabled by using a web-based interface for smart offices. A task or action has a name and often includes parameters as well. Parameters can either represent object names or values. An example of a possible task is *dim-light light32 65* which could represent the users intention to dim light light32 to a light emission percentage of 65.

3.3 REPOSITORY

A repository generally denotes a central place which stores data of components. In the context of the smart office interface, the repository denotes a storage location containing information about objects. The initial state as mentioned in the problem section (3.2.3) is in fact for the largest part generated based on information from the repository. A selection of information present in the repository is:

1. Name of the object (e. g., light_778, comp33)
2. Type of the object (e. g., light, computer)
3. Location of the object (e. g., room54, office21)
4. Most recent value of the object (e. g., 34 Watt, 83%)

3.4 ENVIRONMENT STATE CHANGE NOTIFIER

Since the state of the environment changes often, the (initial) state should be updated. The environment state change notifier is responsible for detecting an update, needs to gather information about the name of the object updated and the new value of the object and informs and updates the repository of every subscriber. Suppose that the interactive whiteboard is turned off due to a long period of inactivity. The environment state change notifier notifies a change in the environment, gathers the name of the whiteboard and the new status (turned-off) and informs and updates the whiteboard object for every subscriber.

3.5 REQUIREMENTS

The functional and non-functional requirements are concisely listed below:

3.5.1 *Functional Requirements*

- The user should be able to register for a new user account
- The user should be able to login with an existing user account
- The tasks should be filtered on the location specified in the user profile
- The state of the environment should be displayed
- The environmental state should be updated if necessary
- Basic information from the user profile should be viewable
- The user should be able to add/remove tasks
- The user should be able to execute tasks
- The result of task execution should be displayed
- Speech input as well as usual (keyboard/mouse) input should be supported

3.5.2 *Non-functional Requirements*

- The interface should be intuitive and easy to use
- There should be a trade-off between functionality and simplicity of the interface
- The interface must be able to work with multiple devices (e. g., desktop computers, laptops, smartphones etc.) having different operating systems
- The interface should work properly on at least the web browsers Mozilla Firefox and Google Chrome.
- The user's attention should "remain focused on the work being done, rather than on the mechanics of interaction" [13]
- The user should learn and remember a limited number of actions [13]

3.6 INTERFACE DESIGN

Designing an interface is a very challenging task and requires a lot of hard work. Knowledge of best practices in smart environments and interface design principles should be combined in order to create a good interface. To support the justification of design decisions, screen shots of the interface are used.

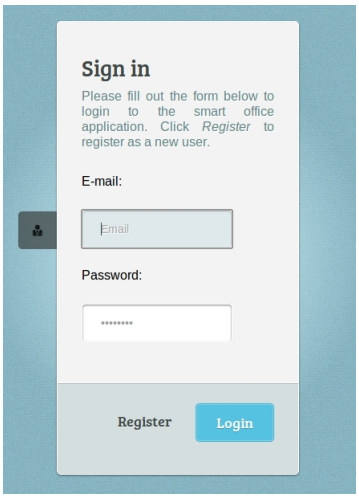

Figure 5: The start screen of the interface.

The **start screen** does not need much explanation since it is very straightforward. The Cascading Style Sheet (CSS) styling is designed by Jakhu. Adjustments to the original design have been made, e. g., the names of the input fields are shown. The size of the login box is relatively small. This design decision has been made with regard to the different screen sizes of different devices. Especially in the case of smart phones, screens are often quite small and this design compensates for such a screen. In the case the user is not registered yet, the user clicks the "Register" button and ends up with the **register screen**. The register screen is only partly shown in Figure 6.

Notice that not all information from the register screen is shown for clarity purposes. Besides giving a valid e-mail address and password (which should be confirmed), the user should also specify the location of the office he/she is working in. The location information is used to determine appropriate objects for the user.

The implementation of the design of the interface (Figure 7) is for the largest part performed by Derby Webdesign [21]. The template has been adjusted to meet the needs of the smart office interface. The **main screen** of the interface contains a welcome message giving a

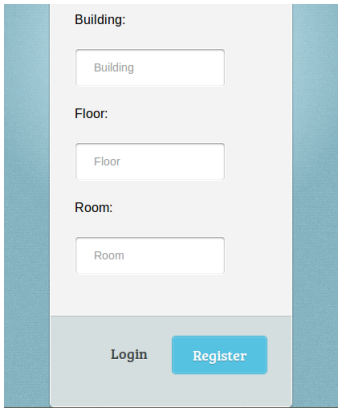

Figure 6: An extraction from the register screen of the interface.

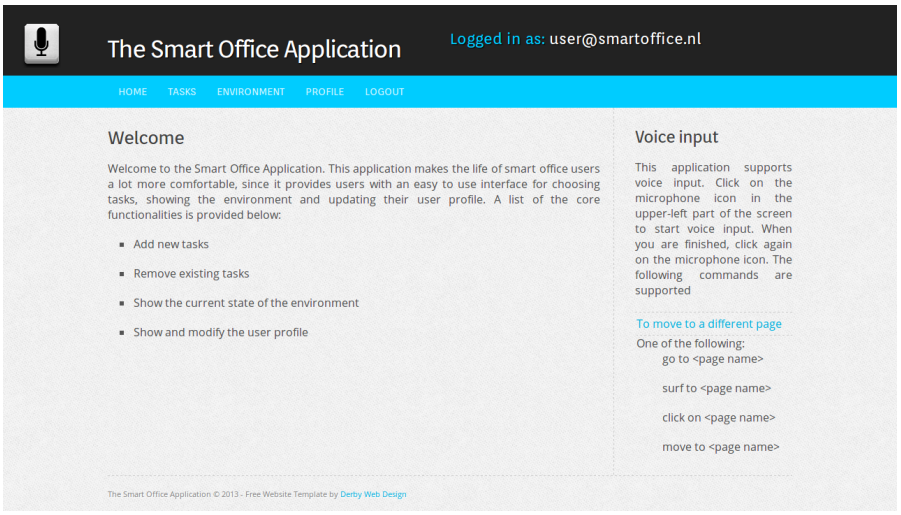

Figure 7: The home screen of the interface.

summary of the functionalities the interface has to offer. On the right-hand side, the user becomes aware of the possibility to make use of voice input. The idea behind this decision is that the user should be aware of the possibilities as soon as possible even if the user will not use the functionality. The possible statements are listed as well, because the user needs to know which voice input it should use to accomplish its desired action. The same action is achieved by more than one sentence which makes sure the application is flexible enough for multiple users having different mindsets. The upper-left of the screen shows a microphone icon: the interface of the WAMI-toolkit. The toolkit is described in section 4.3 in the Implementation chapter.

The main job of the interface is to support **task** selection, the screen dealing with this functionality is displayed in Figure 8:

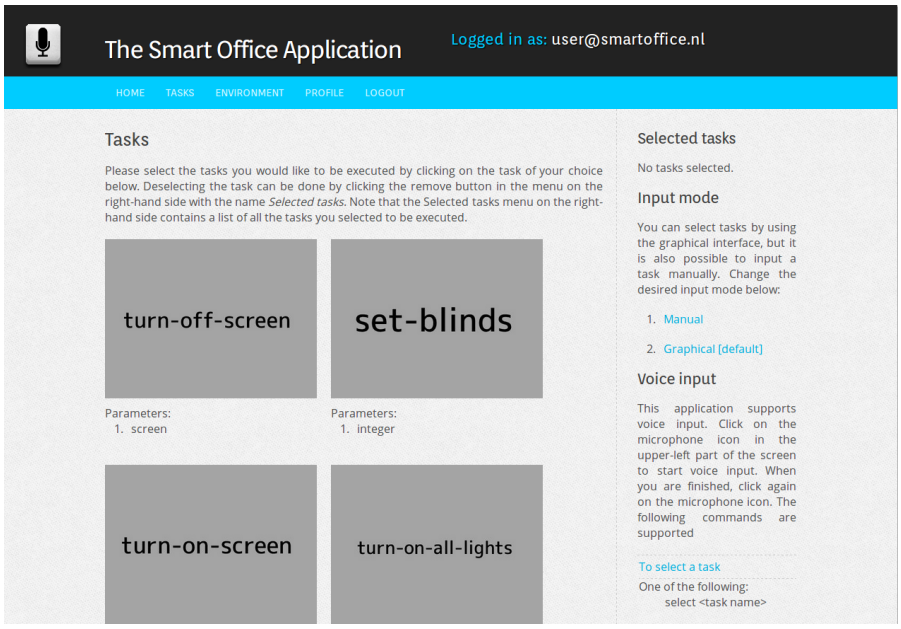

Figure 8: The tasks screen of the interface.

The tasks are graphically displayed by using rectangles with the task names inside. It is difficult to generate useful images based on the task's name. Therefore the application generates images with the help of dummy images [10]. The most ideal scenario would be to display images representing the tasks, but the current system is not ready for such a feature.

Figure 9 shows the screen loaded after the user selected a task. The screen shows the necessary parameters in a well-styled table. The selected object should be applied before going to the next screen, because the application needs to calculate the allowed values in the case an additional value is required (e. g., dimming a light requires the amount of dimming). Screen 10 is shown in this case.

Specifying values is only required for tasks requiring this, so in many scenarios this screen will not be necessary at all. As is the case with many other screens of the application, a user instruction message just beneath the page heading helps the user in performing the right steps to reach a goal. The *specify values* screen lists the selected objects and their minimum, maximum and step values. The user can use the up and down arrows in the text-box on the right-hand side to select a suitable value. Using only these arrows makes sure the value in the textbox conforms to the minimum, maximum and step constraints. This construction has another advantage: error handling can be limited, because values are already constrained. However, the user is allowed to input a value himself/herself (without using the arrows)

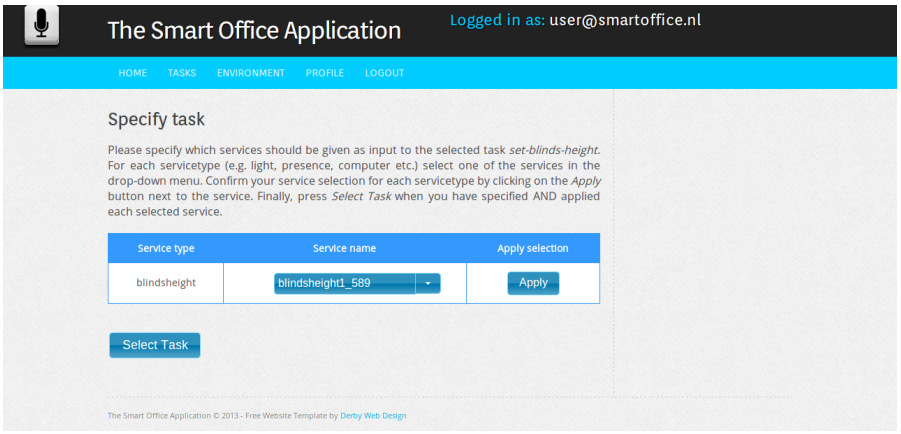

Figure 9: The specify task screen of the interface.

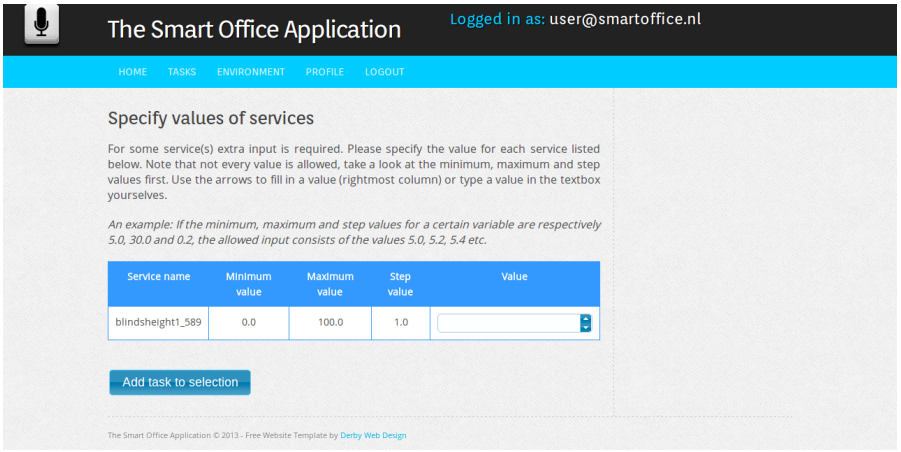

Figure 10: The *specify values* screen of the interface.

and thus can choose an inappropriate value. The interface deals with this situation by displaying warning messages to the user. Clicking the button *Select Task* adds the task to the selected tasks menu as is displayed in Figure 11.

The environment page shows all the available objects (filtered on the user's location) grouped on their type. Figure 12 shows an extraction of this screen for the object type light.

The last screen which is designed is the profile screen. As can be seen in Figure 13 the information provided is very limited. This decision has been made to make sure only the relevant information is shown and to keep the interface as simple as possible.

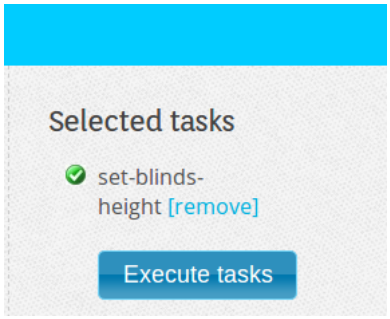

Figure 11: The selected tasks menu which is displayed on the right of the screen.

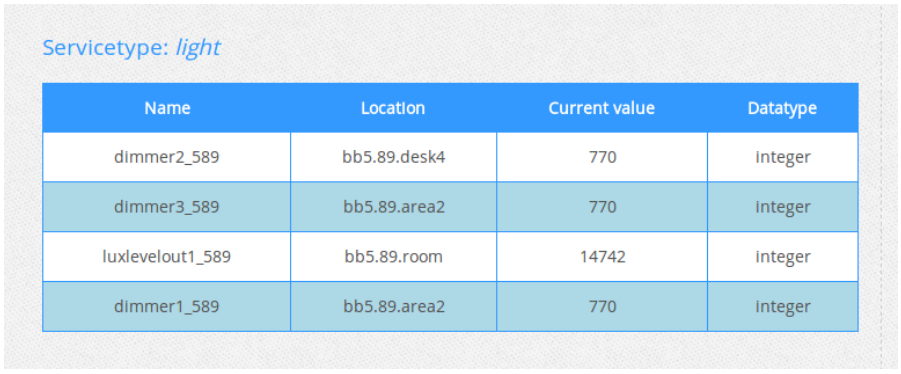

Figure 12: An extraction of the environments page: all the available lights.

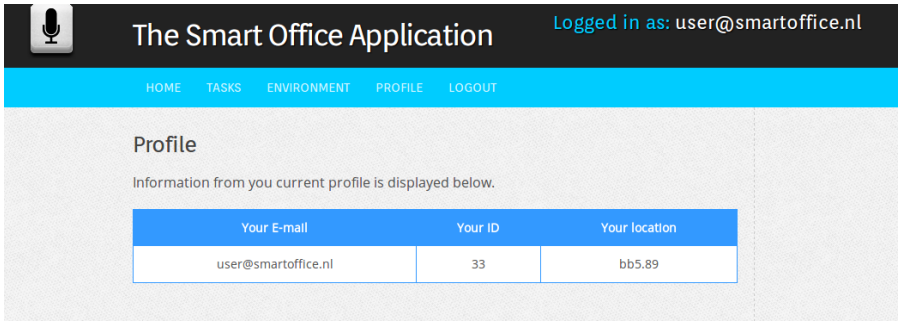

Figure 13: The profile screen of the interface.

3.7 APPLICATION DEPENDENCIES

The web-based application depends on many existing components. An example of such a component is the planner which decides the steps needed to accomplish a user-defined task. The location of the planner needs to be known in order to execute tasks. The decision has been made to store the address of the planner as a constant in the application.

In section 3.3, the information stored in the repository was summed up. The location of the object, which is an example of repository data, needs to have a predefined format because initial state information has to be extracted from it. The initial state information is needed for a proper problem specification. Initial state information also includes data about available locations and their types (refer to section 3.2.3). Suppose there exists a object with location *nijenborgh.dep16.office34*. The application needs to extract from this location the following information:

1. The name of the building
2. The name of the department
3. The name of the office

The names of the building, department and office are respectively *nijenborgh*, *dep16* and *office34*. This information can be extracted due to the predefined format, namely <building >.<department >.<office >.

4

IMPLEMENTATION

Now that the concepts and design have been specified in detail it is time to bring the ideas into practice. The techniques used are elaborated upon and the realization of the concepts introduced in chapter 3 is described.

4.1 FRAMEWORK

Choosing a framework is not an obligatory step, but is often useful especially for developing web-applications. Many web-application frameworks exist, each has a different vision, but they share enabling easy development of a web application. Based on previous experiences the Play Framework [5] was chosen. This framework focuses on programming languages Java and Scala. The framework is "based on a lightweight, stateless, web-friendly architecture and features predictable and minimal resource consumption (CPU, memory, threads) for highly- scalable applications" [4]. The Play Framework uses the Model-View-Controller (MVC) design pattern which divides the design of an application into three parts: data model (model), data representation (view) and application logics (controller). This separation improves the readability and reusage of code. The author used Play Framework version 2.0.4 even though this was not the most recent version at the beginning of the project. This decision has been made in order to make sure the version used is stable enough to develop a web application without (major) flaws. The problem with bleeding edge software, as would be the case when choosing e. g., version 2.1.0, is that it often contains unresolved bugs and that it is not widely adopted by its users yet.

4.2 WEB DESIGN LANGUAGES

The implementation of a web application requires several web programming languages. The languages used are briefly described in this section.

HTML

HyperText Markup Language (HTML) does not need much explanation because of its popularity nowadays. Building a web page without HTML is practically impossible. The Play Framework combines Scala and HTML for creating web pages. This way, it is possible to display application data on a webpage.

CSS

The styling of webpages is done by using CSS for external stylesheets. Using external stylesheets is advantageous because the content of the webpages is separated from the design.

jQuery

A good user interface enables user interaction. The web application uses jQuery for enabling interactive design of webpages.

4.3 WAMI TOOLKIT

A multi-modal interface provides the user with the ability to use multiple modes of interfacing with the system. A multi-modal interface aims at providing more flexibility for the user. Using e. g., speech gives the user the possibility to use more natural ways of interfacing with the application. The WAMI toolkit [9] is suitable for use in this application, because it is very easy to include it on a web application and, most importantly, adds an extra dimension to the application. Several existing multi-modal interfaces require the use of special web browsers, the WAMI toolkit can be used on standard web browsers such as Mozilla Firefox, Opera, Safari and Internet Explorer. A disadvantage of the toolkit is that it depends on Adobe Flash and the user should grant permission for accessing the microphone. Securing the user's privacy is definitely a good thing. On the contrary, the user needs to do more work to use the voice input functionality. In summary, the advantages of the toolkit outweigh the disadvantages so it is a valuable addition to the application.

4.4 PLANNER

Recall from section 3.2 that the planner is responsible for making decisions in the case the user wants a task to be executed. The planner receives a specification of the task and the initial state and generates a list of steps to be taken in order to execute the task. The implementation of the planner is not in the scope of this project: the planner was available at the start of this project.

The application connects to the planner using a RESTful web service. A RESTful web service is a client-server application based on the REpresentational State Transfer (REST) architecture. Every piece of information in a RESTful web service should be referred to as a resource. The RESTful web service allows the application (REST-client) to communicate with the planner (REST-server) over the internet. In fact, the planner receives a large string containing the user-defined problem specification. The Play Framework provides support for interfacing with RESTful web services, so connecting to the planner was relatively straightforward.

The planner used for the development of the application was developed by Georgievski. The HTN planner is named the Scalable Hierarchical Planner (SHP) and its first prototype is fully implemented in Scala. The planner uses depth-first search to find a solution, if there is any. As is the case with other planners, the SHP requires a problem and domain specification. These specifications are implemented using the description language Hierarchical Planning Domain Language (HPDL).

HPDL was an attempt by Georgievski to standardize description languages for HTN planners. HPDL is a task-centred language where a task can be either an action or a method. The language is based on Planning Domain Definition Language (PDDL) which was an attempt by Mcdermott et al. to create a universal language for planners. The extension to HPDL is needed, because PDDL is limited with respect to the possible tasks the user can choose from: only actions can be selected. The strength of the application is the possibility to choose (complex) tasks, thus the planner needs HPDL as the description language. Examples of domains and problems implemented using HPDL [6] are given in the Appendix and are clarified in the following sections.

4.4.1 *Domain*

Figure 3 presented the flow of data between existing systems. Domain knowledge is needed by the planner in order to make decisions. The domain is implemented as a plain-text file structured according to HPDL. An example heading of a domain file is displayed in Listing 1 of Appendix A.

Note that this file is not complete in the sense the allowable actions and tasks are not listed. The structure of the actions and task is discussed later, first the domain heading is explained.

The name of the domain is specified on line 2. Line 3 enumerates the requirements for the planner. The planner uses the requirements to determine if it can handle the domain. If this is not the case, the planner can skip over the domain immediately [6].

Lines 7 - 9 specify the possible types of variables used in action and task definitions. The types are needed to bind appropriate objects to task/action parameters. In this domain, a variable can be of type floor, room, light or screen.

A predicate is a statement that is either true or false. The planner needs the predicates to know the possible statements in the domain. The predicates in this example can be used to state that a room is *on* a floor, something is *in* a room, a device is *turned-on*, a light has a value (the level of lux) and a sensor has a value. The name immediately following the opening bracket indicates the name of the predicate and the following words represent variables of the predicate. In HPDL words starting with a question mark denote a variable.

A different example which deals with an action is given in Listing 2 of Appendix A. The domain heading is omitted since it is already explained above.

Recall that an action is a simple task with an observable effect. *turn-of-screen* is an action which has parameters of types floor, room and screen. Of course the planner needs to know which screen should be turned off and in which room and floor the screen is located. Most actions can only be executed when the environment complies with certain conditions. These conditions are specified in the *precondition*. The precondition states that in order to execute the action *turn-of-screen*, the room should be located *in* the floor, the screen should be located in the room and the screen should be turned on. It does not make sense to turn off a screen while it is already off and that justifies the last condition. The effect, also called postcondition, explains what will change in the environment after the action has been executed. In

this example, the effect is very simple because the only effect is that the screen parameter is not turned on, exactly what the action name is indicating.

The last example listed in 3 shows a domain file handling a task which generally consists of multiple methods.

The task used in the example is *print-a-document* which is a realistic example of a daily pursuit of an office user. As described in section 3.2.1.1, a method is a possible way to achieve a task. A method can distinguish different states of the environment which is the case in listing 3. The first method handles the case in which the printer is turned off. The precondition needs to be satisfied, so the room should be located *in* the floor, the printer should be located *in* the room and the printer should, of course, be turned off. The actions taken to print the document will be to turn the selected printer on first and then to send the document to the printer. The *sequence* keyword is used to emphasize the importance of the order of execution. Interchanging the actions *turn-on-printer* and *print-document* will have dramatic effects, because the printer is not turned on and the document will probably not be printed at all. In the case the order of execution is not important, the keyword *unordered* should be used instead of sequence. The second method is straightforward since the printer is already turned on and the only thing which needs to happen is to send the document to the printer. Notice the use of the keyword unordered here, even though in this case we could have used sequence as well, because there is only one action to be executed.

4.4.2 *Problem*

Domain knowledge is not enough to get the planner started, because the planner needs to know which decision-making problem needs to be solved. The first part of an example problem file specified in HPDL is given in Listing 4 of Appendix B.

Note the correspondence of the structure of the *init* part and the structure described in section 3.2.3. The first two lines in the init part fit in rule 1 represent the fact that firstFl is a floor and room848 is a room. The three lines following are examples of rule 2. They state that deskLight02 and deskLight03 are lights and allInOnePr12 is a printer. The next two lines comply with rule 3 and indicate that deskLight03 has value 340 (this value could for instance be the amount of illuminance) and that allInOnePr12 is turned on. Finally, the last two lines

illustrate rule 4: room848 is located on floor firstFl and allInOnePr12 is in room848.

The next example, listed in 5 of Appendix B, illustrates the structure of the task specification.

A specification of a task can consist of any combination of actions and tasks, therefore it is allowed to execute multiple tasks/actions at once. This powerful feature of the application is illustrated in Listing 5. The planner should figure out a plan which makes sure that light deskLight02 is turned on first. Document unnamedDoc should then be printed on printer allInOnePr12 (located on floor firstFl in room room848). Finally, screen76 should be turned off.

The problem file needs to be generated by the application. This is in contrast to the domain file which is already present and is used by the application. Information about the initial state is gathered from the repository (4.5) whilst task specification is done by the user of the smart office interface. A domain file on its own is not useful for the planner, a problem file is not either. The combination of domain and problem file defines a problem with respect to a certain domain and this decision-making problem can be handed over to the planner.

4.5 REPOSITORY

The repository stores data about services. A service is defined as an object of a specific type located somewhere. The repository is implemented as a Cassandra-based database [1] system. The relevant information for the smart office interface can be extracted from it as a JavaScript Object Notation (JSON)-file. This file is not generated by the application but was already available at the start of this project. An example repository implementation is given in Listing 6 in Appendix C.

The squared brackets indicate that a list is declared. Curved brackets represent an object. An object has several fields, separated by a comma. Each field has a key and a value. For example, line 3 specifies a field with key "servicetype" and value "thermostat". Example applications of the service data include:

- Representing the available services in a given location
- Selecting task parameters for task execution
- Limiting the possible values for a task parameter

- Generating domain (initial state) knowledge: location of a service

4.6 ENVIRONMENT STATE CHANGE NOTIFIER

The environment state change notifier is simulated using RabbitMQ [18]. RabbitMQ is a message broker which enables messaging for the smart office interface. RabbitMQ can be used to send state updates to the application by using e. g., the producer-consumer mechanism in combination with an exchange. Suppose the value of service *thermostat3_292* is updated to 19.5. All subscribers are being notified, receive the new value of the corresponding service and update the "updatedvalue" field of the service with "varname" thermostat3_292 to 19.5.

4.7 APPLICATION ARCHITECTURE

In this section the architecture of the application is graphically displayed and the diagrams are elucidated. Using a top-down approach, the most important classes of the application are explained. Not all fields/methods are shown in the class diagram, only the most important data is shown.

The class **UserData** is the entry point of the application and contains all the data a user needs in order to work with the application. Figure 14 shows this class and its relationship with two other classes **InitObject** and **Task**. UserData contains the field *selectedTaskArgs* which is needed to remember the arguments for a specific selected task. *inputMode* represents the selected input mode for task specification: either manual or graphical (default). In the case the state of the environment has changed, the method `updateData` is used.

An InitObject has a list of **SimpleInitObject**s containing all the available objects. The method `readInitFile` is very important: it makes sure the objects are read and stored. Initial state information is added by using `addLocationAndVariablePredicates`. The Task class basically contains a list of tasks. A task has two important fields: the name of the task and the list with parameter types.

As could be remarked about Figure 14, UserData has two more important fields: *service* and *goal*. Figure 15 graphically displays the relationship of UserData with the corresponding classes. The class **Service** has as main task keeping a list of objects which are grouped on their type. This is needed for specifying task parameters: a param-

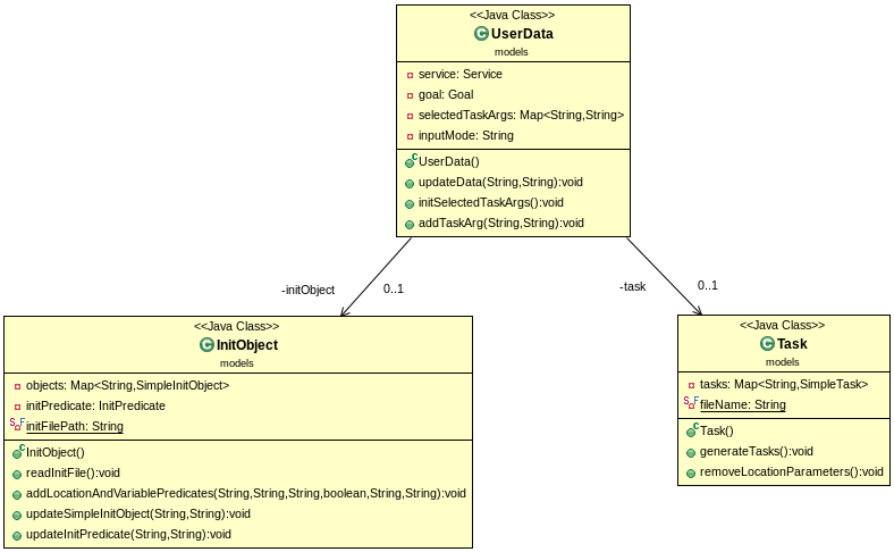

Figure 14: The UserData class and two related classes.

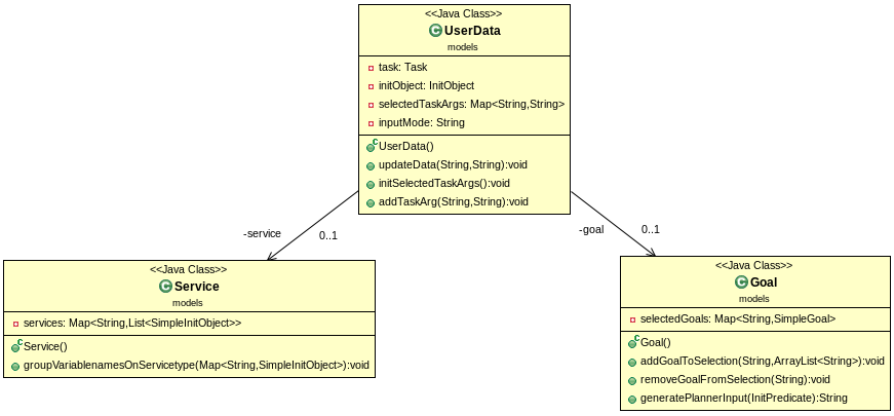

Figure 15: The UserData class and the two other related classes.

eter is of a certain type and thus only the objects with this type are appropriate.

The **Goal** class knows which tasks are selected as goal and provides methods to add or remove tasks. If the user wants to execute tasks, the `generatePlannerInput` method is called to prepare the input for the planner.

5

CONCLUSIONS AND FUTURE WORK

The presented work proposes a web-based interface for smart offices in order to make the role of the smart office user more active. The web application was developed taking into account several objectives. Each objective is restated and contains a description including whether the objective is accomplished and what the future work is that should be done.

1. **Make the role of the smart office user more active: the smart office autonomously makes decisions which do not always satisfy the user's needs**

   By using the smart office interface, the author claims the user's role has become more active due to giving the user the ability to select (complex) tasks. Furthermore, the user can request the status of the environment, so the user is more aware of what is going on in the environment. The combination of the smart office autonomously making decisions without requiring user involvement and the user defining what should happen in the environment makes the life of the smart office user more comfortable: On the one hand many things are regulated automatically, but on the other hand the user has the possibility to influence the actions taken in the office.

2. **Create a user-friendly interface which is unintrusive and easy to use**

   User-friendliness, unintrusiveness and easiness of use are subjective requirements which need to be evaluated by the user. In order to establish realizable results, the smart office interface should be judged by the user. Therefore, a suggestion for future work would be to conduct an investigation focusing on the usability of the smart office interface. Nevertheless, during the development of the interface care has been taken to make the interface as user-friendly as possible, so that the user regards the interface as unintrusive and easy to use.

3. **Provide the ability to select complex tasks and to show the environmental state**

   The interface was mainly developed to enable the user to select complex tasks. The prototype shipped with this paper includes a graphical and manual interface for selecting and executing tasks of any complexity, so the first part of the objective is accomplished. The second part of the objective is also accomplished, because the smart office interface contains an environment page on which all the available objects are grouped on their type.

4. **Enable easy and natural interaction such as speech-based interaction**

   Even though many users are comfortable with using mouse and keyboard as input devices, for smart offices more natural ways of interaction are highly desired. The web application enables voice input by making use of the WAMI toolkit (as described in more detail in section 4.3). The evaluation of this feature by the smart office user has not been performed and can be considered as future work. Irrespective of the user's judgment, selecting a specific task from a large number of tasks is probably much easier when using voice input compared to using the usual mouse/keyboard combination.

5.1 FUTURE WORK

There are several interesting improvements possible which had to be left aside due to time restrictions.

The first feature that would make the application more user-friendly is to represent tasks by using representative images. The current graphical representation of tasks makes use of automatically generated text-based images. The implementation of representative images requires additional information. Using e. g., useful icons for representing tasks would make the smart office interface more manageable and thus more easy to use. This feature could be added by e. g., storing the icons and their names in a database.

The environment page of the web application groups the available objects on their types. Johanson et al. states that "the interface should take advantage of natural mappings to the physical structure" which can be achieved by sustaining the hierarchy of the environment: e. g., let the user select the building first, then the floor and finally the

room. A different approach to make the environment page more user-friendly is to show images of the devices present in an office: again this requires storing the images in e. g., a database.

The prototype shipped with this paper allows users to select and execute tasks but is not connected to physical devices such as sensors and actuators. Adding this connection enables users of the smart office interface to control an office physically, which is exactly where the interface is necessary for.

The application currently has limited support for voice input. Extension of this functionality will probably lead to a more user-friendly application. A possible extension is the feature to select and execute actions immediately by using one simple voice command.

BIBLIOGRAPHY

[1] Apache. The Apache Cassandra Project Homepage, July 2013. URL http://cassandra.apache.org/. (Cited on page 26.)

[2] Juan Carlos Augusto. Past, present and future of ambient intelligence and smart environments. In *ICAART*, pages 11–18. INSTICC Press, 2009. ISBN 978-989-8111-66-1. URL http://dblp.uni-trier.de/db/conf/icaart/icaart2009.html#Augusto09. (Cited on page 1.)

[3] Diane J. Cook et al. *Smart environments - technologies, protocols and applications.* Wiley, 2nd edition, 2005. ISBN 978-0-471-54448-5. (Cited on page 3.)

[4] Play Framework. Play Framework 2.0.4 documentation, April 2013. URL http://www.playframework.com/documentation/2.0.4/Home. (Cited on page 21.)

[5] Play Framework. Play Framework homepage, April 2013. URL http://www.playframework.com. (Cited on page 21.)

[6] I. Georgievski. Hierarchical Planning Domain Language. Technical Report 0, Johann Bernoulli Institute for Mathematics and Computer Science, 2013. (Cited on pages 23 and 24.)

[7] I. Georgievski. Scalable Hierarchical planner homepage, July 2013. URL http://planner.ilche.info. (Cited on page 23.)

[8] Ilche Georgievski et al. An overview of hierarchical task network planning. Technical report, University of Groningen, JBI 2012-12-5, 2012. (Cited on page 10.)

[9] Alexander Gruenstein et al. The WAMI toolkit for developing, deploying, and evaluating web-accessible multimodal interfaces. In *Proceedings of the 10th International Conference on Multimodal Interfaces (ICMI '08), Chania, Crete, Greece*, pages 141–148, 2008. (Cited on page 22.)

[10] Dummy Images. Dummy Image Generator, April 2013. URL http://www.dummyimage.com. (Cited on page 17.)

[11] Amit Jakhu. Login Form, April 2013. URL http://designerfuel.tumblr.com/post/15555140593/login-form-psd-live. (Cited on page 15.)

[12] Zhang Jianhong et al. Improved htn planning approach for service composition. In *Proceedings of the 2004 IEEE International Conference on Services Computing*, SCC '04, pages 609–612, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2225-4. URL http://dl.acm.org/citation.cfm?id=1025130.1026213. (Cited on page 10.)

[13] Brad Johanson et al. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, 1:67–74, 2002. (Cited on pages 14 and 30.)

[14] Eirini Kaldeli et al. Coordinating the web of services for a smart home. *ACM Trans. Web*, 7(2):10:1–10:40. ISSN 1559-1131. (Cited on pages 10 and 11.)

[15] Tiiu Koskela et al. Evolution towards smart home environments: empirical evaluation of three user interfaces. *Personal Ubiquitous Comput.*, 8(3-4):234–240, July 2004. ISSN 1617-4909. (Cited on pages ix, 4, 5, and 6.)

[16] Christophe Le Gal et al. Smart office: Design of an intelligent environment. *IEEE Intelligent Systems*, 16(4):60–66, July 2001. ISSN 1541-1672. (Cited on page 6.)

[17] D. Mcdermott et al. PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control, 1998. (Cited on page 23.)

[18] RabbitMQ. RabbitMQ homepage, April 2013. URL http://www.rabbitmq.com. (Cited on page 27.)

[19] Carlos Ramos et al. Smart offices and intelligent decision rooms. In *Handbook of Ambient Intelligence and Smart Environments*, pages 851–880. 2010. (Cited on page 6.)

[20] Dirk Roscher et al. A meta user interface to control multimodal interaction in smart environments. In *IUI '09: Proceedingsc of the 13th international conference on Intelligent user interfaces*, pages 481–482, 2009. (Cited on pages ix, 4, and 5.)

[21] Derby Webdesign. Response Template, April 2013. URL http://www.derby-webdesign.co.uk/2012/09/

response-free-responsive-website-template. (Cited on page 15.)

[22] Qiang Yang et al. Learning recursive HTN-method structures for planning. In *Workshop on Artificial Intelligence Planning and Learning, Proceedings*, 2007. URL http://www.cs.umd.edu/~ukuter/icaps07aipl/. (Cited on page 10.)

Part I

APPENDIX

A

EXAMPLE DOMAIN FILES IN HPDL

In this chapter of the Appendix examples of **domain** files specified in HPDL are given.

A.1 DOMAIN HEADING

Listing 1 below shows a first example domain file in HPDL. Three dots are used to indicate that code which can be spread over multiple lines is left out.

Listing 1: An example domain file in HPDL.

```
(define
  (domain smartoffices)
  (:requirements
    :strips :typing :negative-preconditions
    :universal-preconditions
  )
  (:types
    floor room light screen
  )
  (:predicates
    (on ?room ?floor)
    (in ?something ?room)
    (turned-on ?device)
    (light-value ?light ?luxLevel)
    (light-value-sensor ?sensor ?luxLevel)
  )
  ...
  (:action ... )
  ...
  (:task ... )
  ...
)
```

A.2 DOMAIN ACTION

Listing 2 below shows an example domain file illustrating an action. Three dots are used to indicate that code which can be spread over multiple lines is left out.

Listing 2: An example domain file illustrating an action.

```
(define
  ...
  (:action turn-off-screen
    :parameters (?f - floor ?r - room ?s - screen)
    :precondition (and (in ?r ?f) (in ?s ?r) (turned-on ?s))
    :effect (not (turned-on ?s))
  )
  ...
)
```

A.3 DOMAIN TASK & METHOD

Listing 3 below shows an example domain file illustrating a task and two methods. Three dots are used to indicate that code which can be spread over multiple lines is left out.

Listing 3: An example domain file illustrating a task and two methods.

```
(define
  ...
  (:task print-a-document
    :parameters (?f - floor ?r - room ?d - document ?p - printer)
    (:method printer-turned-off
      :precondition (and (in ?r ?f) (in ?p ?r) (turned-off ?p))
      :tasks (sequence (turn-on-printer ?f ?r ?p) (print-document
          ?d ?p))
    )
    (:method printer-turned-on
      :precondition (and (in ?r ?f) (in ?p ?r) (turned-on ?p))
      :tasks (unordered (send-document-to-printer ?d ?p))
    )
       ...
  )
  ...
)
```

B

EXAMPLE PROBLEM FILES IN HPDL

In this chapter of the Appendix examples of **problem** files specified in HPDL are given.

B.1 PROBLEM HEADING

Listing 4 below shows a first example problem file in HPDL. Three dots are used to indicate that code which can be spread over multiple lines is left out.

Listing 4: An example problem file in HPDL.

```
(define
  (problem smartofficesproblem)
  (:domain smartoffices)
  (:requirements :strips)
  (:init
    (floor firstFl)
    (room room848)
    (light deskLight02)
    (light deskLight03)
    (printer allInOnePr12)
    (light-value deskLight03 340)
    (turned-on allInOnePr12)
    (on room848 firstFl)
    (in allInOnePr12 room848)
  )
  (:goal-tasks
    ...
  )
)
```

B.2 PROBLEM TASK

Listing 5 below shows an example problem file illustrating task specification. Three dots are used to indicate that code which can be spread over multiple lines is left out.

Listing 5: An example problem file illustrating task specification

```
(define
  ...
  (:goal-tasks
    (sequence
      (turn-on-light deskLight02)
      (print-a-document firstFl room848 unnamedDoc allInOnePr12)
      (turn-off-screen screen76)
    )
  )
  ...
)
```

C

EXAMPLE REPOSITORY FILE

An example repository file is given below. Three dots are used to indicate that more objects are following.

Listing 6: An example extraction from a repository file in JSON.

```
[
  {
    "varname":      "thermostat3_292",
    "servicetype":   "thermostat",
    "varlocation":   "nb.92.room",
    "updatedvalue": "15.0",
    "controllable": "true",
    "datatype":      "float",
    "domainstates":
    {
       "range" :
       {
          "max":   "30.0",
          "min":   "5.0",
          "step": "0.1"
       }
    },
    "routingkey": "nijenborgh.floor2.room292.actuators.
        thermostats.thermostat3_292",
  },
  ...
]
```

D

INSTALLATION GUIDE

Running the application is straightforward, because the Play Framework has built-in functionalities to generate a stand-alone executable. The various steps needed to get the application up and running are explained in detail below.

D.1 INSTALLATION INSTRUCTIONS

The steps needed to install the application are described below:

1. Extract the attached .zip file to a location of your choice.
2. Start a terminal of your choice and change your current directory to the location of the .zip file you extracted in step 1.
3. Run the file *start* in the terminal.
4. Start Google Chrome/Chromium or Mozilla Firefox.
5. Surf to *http://localhost:9000* and you are ready to use the web-application.

D.1.1 *Clarification*

Extract the attached .zip file to a location of your choice
Most Operating Systems have built-in unzip applications. If this is not the case, you can use e. g., 7-zip (http://www.7-zip.org/) which works on several operating systems.

Start a terminal of your choice and change your current directory to the location of the .zip file you extracted in step 1
You might need to change the access settings of the *start* file. For Linux-like systems, "chmod +x ./start" will probably work.

Start Google Chrome/Chromium or Mozilla Firefox
The application is thoroughly tested on Google Chrome/Chromium and Mozilla Firefox. Other browsers can be used, but this is not recommended. Especially the layout of the webpages may deviate.

Surf to *http://localhost:9000* and you are ready to use the web-application
Hint: store the URL in your bookmarks to be able to access the application faster.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

DECLARATION

I hereby declare that this thesis is my own work and effort and that it is not submitted anywhere else for any award. Usage of external sources is acknowledged where applicable.

Groningen, July 2013

Jorrit de Boer