UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE
TELECOMUNICACIÓN

A Model for Mobile-Instrumented Interaction and Object Orchestration in Smart Environments

Tesis Doctoral

LUCA BERGESIO

Ingeniero en Informática

Máster en Tecnologías y Sistemas de Comunicaciones

2017

Departamento de Señales, Sistemas y Radiocomunicaciones

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE
TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE MADRID

A Model for Mobile-Instrumented Interaction and Object Orchestration in Smart Environments

Tesis Doctoral

Autor:

LUCA BERGESIO

Ingeniero en Informática

Máster en Tecnologías y Sistemas de Comunicaciones

Directores:

ANA MARÍA BERNARDOS BARBOLLA

Doctora Ingeniera de Telecomunicación

JOSÉ RAMÓN CASAR CORREDERA

Doctor Ingeniero de Telecomunicación

June 28, 2017

“The jump from zero to whatever baud rate is the most important jump you can make. After that everyone always wants to go straight to the speed of light”

J. T. Tengdin’s First (and Only) Law of Telecommunications

“When you see a good move, look for a better one”

Emanuel Lasker, World Chess Champion from 1894 to 1921

Abstract

The proliferation of the smartphones has given a considerable boost to the spread of the smart objects and the consequent creation of smart spaces.

Smart objects are electronic devices that are able to work interactively and autonomously, usually preserving the interaction metaphor of their non-electronic counterpart. Through a network interface, they can cooperate with other objects: their strengths do not lie in their hardware, but in the capabilities to manage interactions among them and in the resulting orchestrated behaviour. Smart spaces are environments composed of smart devices, where they work together, producing some behaviour of benefit to the dwellers.

The current workflow requires that the user downloads an application from a digital distribution platform for each smart object he wants to use. This model converts the smartphone in a control centre, but it limits the potentialities of the objects. Devices are connected in a one-to-one network with the smartphone, a configuration that prevents the direct communication among the objects and that puts the responsibility to coordinate the objects in the hands of the smartphone. Moreover, there are only a few frameworks that permit the integration of several applications and the creation of complex behaviours that involve many objects from different manufacturers.

The first challenge considered in this thesis is to propose a new workflow that permits to integrate any kind of smart device in any behaviour of the smart space. The workflow will include the discovery of the new objects and their configuration, without the need of downloading a new standalone application for every object. It will provide to the user a simple configuration tool to create personalized behaviours (scenes), based on the event-condition-action paradigm. Finally, it will automatically orchestrate the smart devices to produce the desired behaviours of the environment.

Smart spaces are thought to behave in a personalized way, adapting to the particularities of their inhabitants. “Personalization” is about understanding the needs of each individual and helping satisfy a goal that efficiently and knowledgeably addresses each individual’s need in a given context. Thus, the second challenge tackled in this thesis is to move forward on how to evolve smart spaces from customizable

to personalized environments.

The third open issue considered in this research is how to make portable the personalized configuration, i.e. how to enable a user to commute among different smart spaces preserving and adapting his personalized settings to each of them.

Both personalization and portability will be included in the tool to automatically aid the user to gain a full and transparent control over the environment.

Solutions to tackle device fragmentation, interoperability, seamless discovery, scene modelling, orchestration and reasoning are needed to achieve these goals. In this context, the contributions of the thesis can be summarized as follows:

- The definition of a workflow that permits to personalize and control a smart space using the event-condition-action paradigm.
- The design of a model to describe any kind of smart object and its capabilities.
- The extension of the previous model to describe a smart space as a composition of its smart objects.
- The application of the model to the workflow, reinterpreting the main Object-Oriented Programming features and using them to describe the interactions between objects and the recommendation process.
- The proposal of an architecture that implements each step of the workflow and its relations with the model.
- The proposal, development and evaluation of service concepts in real smart space settings.

Contents

Abstract	v
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Context and Motivation	3
1.2 Contributions	6
1.3 Document Structure	7
2 State of the Art	9
2.1 Smart Spaces and Smart Objects	9
2.2 Personalization in Smart Spaces	12
2.3 Smart Spaces Modelling	14
2.4 Architectures for Smart Spaces	16
2.4.1 Interoperability	18
2.4.2 Discovery	20
2.5 Mobile-Mediated Interaction in Smart Spaces	22
2.6 Summary	25
3 Operational Concepts and Workflow	29
3.0.1 A Scenario	29
3.1 Operational Concepts	31
3.1.1 Identification and Location	34
3.2 System Workflow	37
3.3 Technology Agnostic Architecture	41
3.4 Summary	44
4 Smart Object Modelling	45
4.1 Using OOP to Model and Network Smart Objects	47

4.2	Smart Object Model Proposal	48
4.2.1	Modelling a Smart Object	53
4.3	Formalization of the Generic Object Model	59
4.3.1	Modelling Sensors	64
4.3.2	Modelling Actuators	67
4.3.3	Events, Conditions and Actions in the Model	70
4.3.4	Modelling a Smart Object	72
4.4	Summary	80
5	An Architecture for Smart Spaces Using the OOP Smart Object Model	83
5.1	Architecture Overview	83
5.1.1	Scene Description	85
5.2	Module Content	89
5.3	Discovery of Smart Objects	92
5.4	Personalization of the Smart Space	96
5.4.1	Manual Scene Creation	97
5.4.2	Recommendation	99
5.4.3	Generalization and Specialization	100
5.4.4	Case-Based Reasoner	107
5.4.5	Adaptation of previous scenes	111
5.4.6	Scene Compatibility Analysis and Object Configuration	113
5.5	Smart Space Orchestration	114
5.6	Summary	116
6	Service-Oriented Validation	119
6.1	Experimental Scenario	119
6.2	MECCANO	120
6.2.1	Smart Objects	121
6.2.2	Service Discussion	123
6.3	PERSEO	128
6.3.1	Smart Objects	130
6.3.2	Service Discussion	131
6.4	Follow-Me	135
6.4.1	Smart Objects	137
6.4.2	Service Discussion	137
6.5	Air Traffic Control Station	139
6.5.1	Service Discussion	140
6.6	Summary	142

7	Conclusions and Future Research Directions	145
7.1	Contribution Summary	145
7.2	Future Work	147
7.2.1	Model	147
7.2.2	Standardization	147
7.2.3	Case-Based Reasoning	147
7.2.4	Security	148
7.2.5	Shared Objects	148
7.2.6	Scalability	149
7.2.7	Interaction Issues	149
7.2.8	Coding	149
A	Architecture of the Orchestrator	151
A.1	Scene Description	154
B	Creation of a Smart Object	157
B.1	Model	157
B.2	Hardware	158
B.3	Arduino Code	158
B.4	Notification Server	161
B.5	Module	162
B.6	Discovery	167
	References	169

List of Figures

2.1	Some commercial smart objects	11
2.2	Some smart home projects	12
2.3	Three kinds of semi-automatic discovery used in commercial devices	22
3.1	System workflow	37
3.2	Technology agnostic architecture	42
4.1	Components of a generic smart object	52
4.2	Model of a video wall	54
4.3	Model of a smart door	56
4.4	Model of a drone with camera	58
4.5	Generic smart object described with a class diagram	60
4.6	Extension of the Sensor class	65
4.7	Model of a LM35 temperature sensor	66
4.8	Extension of the Actuator class	68
4.9	Model of a LIFX Color 1000's LED bulb	69
4.10	Video wall class diagram	73
4.11	Video wall manifest	74
4.12	Smart door class diagram	76
4.13	Smart door manifest	77
4.14	Drone class diagram	79
4.15	Drone manifest	80
5.1	Architecture overview	84
5.2	Scene representation with class diagram	87
5.3	Example of a module.	90
5.4	First steps of the discovery phase.	93
5.5	Discovery blocks working together to filter capabilities	95
5.6	Example of scene creation	98
5.7	Adaptation of a scene through generalization and specialization	100
5.8	Reference for the variable names used in the algorithms	103

5.9	The CBR cycle	108
5.10	Example of three proposed scenes	112
6.1	Three pictures of the Experience Lab.	120
6.2	Hall of the lab with the smart door and the smart bulb.	121
6.3	Model of the smart door used in MECCANO service.	122
6.4	Model of the smart work lamp used in MECCANO service.	122
6.5	Model of the smart wall spotlight used in MECCANO service.	122
6.6	Scene representation with class diagram	125
6.7	Generalization of the proposed scene	126
6.8	Adaptation of the general scene	127
6.9	PERSEO workflow.	129
6.10	PERSEO video service.	133
6.11	Generalized video service.	134
6.12	Follow-Me service.	136
6.13	Prototype of the ATC station.	140
A.1	Architecture overview	152
B.1	Model of the smart door.	157
B.2	Hardware used to create the smart door.	159
B.3	NFC sticker attached to the door.	168

List of Tables

4.1	Definition and reinterpretation of the main OOP features	49
4.2	Components of a smart object	53
4.3	Definition and example of implementation of <code>getValue()</code> along the LM35's inheritance chain.	67
4.4	Workflow markers used in the manifests	72

Chapter 1

Introduction

During Palaeolithic Age, humans began to develop the most primitive stone tools: the first man-made objects. Soon, many tools became of common use, until converting some of them into everyday objects. That period marked the beginning of the human technology.

The relation between human beings, technology and objects is thus long standing. Everyday objects have established purposes and have played vital roles in our everyday lives, becoming a daily physical interface between ourselves and the world. The recent growth of computing technologies enables us to reconsider the roles of these objects and in fact poses interesting research and design opportunities to instrument them in a way that radically enhances their well-established features.

The term ‘ubiquitous computing’ was introduced by Mark Weiser in the early 1990s. He described a scenario in which people interacted with and used computers without thinking about them: “*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. [...] Therefore we are trying to conceive a new way of thinking about computers in the world, one that takes into account the natural human environment and allows the computers themselves to vanish into the background.*” (Weiser, 1991). Following the ubiquitous computing concept, in 1995, Weiser also described the calm technology as “*that which informs but doesn’t demand our focus or attention*” (Weiser and Brown, 1996): a type of information technology where the interaction between the technology and its user is designed to occur in the user’s periphery rather than constantly at the centre of attention.

Nowadays Weiser’s ‘computers’ are embedded physical devices, such as household appliances, watches or thermostats. They are equipped with microprocessors and network transceivers, offering communication features and providing smart capabilities. These devices are known as smart objects and their goal is to facilitate

communication and interaction among the environment, other objects and the people. The spread of these devices was estimated at 3.75 billion in 2014, 8.38 billion in 2017 and it is expected to be 20.4 billion of things by 2020 ([Gartner Inc., 2017](#)). Their strengths do not lie in their hardware, but in the capabilities to manage interactions among them and in the resulting orchestrated behaviour. Often the user only perceives their behaviour following the calm technology principle. The communication capability is probably the key of their spread. It was conceptualized in 1968 as Machine to Machine (M2M) and evolved first into Device to Device (D2D) and finally into the Internet of Things (IoT). IoT is *“the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment and the confluence of efficient wireless protocols, improved sensors and cheaper processors”* ([Vermesan and Friess, 2014](#)). IoT is expected to usher in automation in nearly all fields. One of them, also derived from ubiquitous computing and calm technology, are the smart environments: *“small worlds where different kinds of smart device are continuously working to make inhabitants’ lives more comfortable”* ([Cook and Das, 2005](#)). Smart homes are an extension of calm technology due to their emphasis on awareness and adaptability to the user’s needs ([Rogers, 2006](#)).

IoT is the biggest promise of the technology today, but a number of challenges need to be overcome in order for IoT to achieve its objective. In the literature, it is possible to find different groups of challenges, but four of them are common in most publications: security, connectivity, energy and standardization ([Sadiku et al., 2016](#)). The first is security. People’s concern is that the IoT is being developed rapidly without due consideration of the profound security challenges involved. Security is an essential pillar of the Internet and one that the Internet society perceives to be equally essential and the most significant challenge for the IoT. From security, other important ‘minor’ challenges emerge, one of them is the data privacy. The second challenge is the connectivity. IoT devices can use different communication technologies and the connection to the same global network is still a problem. This challenge is also extended to the protocol level. The supply of IPv4 addresses was exhausted in 2014. The deployment of IPv6 will solve the problem, but this process is slow in the Internet and it also affects the IoT that must maintain IPv4 compatibility to be able to work. In a computer both standards can coexist, but often in small devices it is not possible. The third challenge is the power consumption of the IoT devices. They should work over batteries for prolonged periods of time. Increasing the capacity of the batteries and optimizing the applications for low energy consumption is one of the challenges. The last point is the standardization. A period of peace during the protocols war may simplify the integration of the devices and increase their longevity.

The behaviour of a smart space should be beyond pure home automation (i.e.

actuator reaction coordinated to sensor events): smart spaces are supposed to behave in a personalized way, taking into account the particular characteristics of their different inhabitants. For a system or a service, personalization is about “*understanding the needs of each individual and helping satisfy a goal that efficiently and knowledgeably addresses each individual’s need in a given context*”, as Riecken states (Riecken, 2000). Personalization processes usually initiate with some kind of customization input, in which the user configures the system according to his preferences (Braynov, 2003). How to evolve from customizable to personalizable smart spaces is still an open challenge. Additionally, it is also open how to make portable the personalization achieved in different contexts, i.e. how to enable a user to move through different smart spaces preserving and adapting the personalization knowledge to each environment.

1.1 Context and Motivation

In recent years there has been a massive spread of smart objects in several areas and home automation is one of them. According to recent surveys, smart objects for home automation are currently the second best-selling class of these devices, only behind the wearable objects, but it will be the first in the next five years. This trend is driving the research to new technologies and the product development toward that field. In any case, the integration of several different types of smart objects is key to create a functional smart space that supports its occupants.

Nowadays smart objects are mainly installed and configured one by one, a slow process that often does not allow the user to take into account all the available devices and all the possible interactions among them. With the future improvement of the objects it will be increasingly difficult to take advantage of all the possibilities that they will provide us. Today I can buy a smart light bulb, I can remotely switch it on or off or change its colour or brightness. Then I can configure the bulb to switch on and off at a pre-set time, so I do not need to control it manually. Unfortunately the amount of daylight varies throughout the year, so I can enhance my bulb’s behaviour using a home weather station that has a light sensor. In any case, it is useless to switch the light on if nobody is in the room, so I can further enhance my smart light service adding a presence sensor that switches on the bulb only if someone is in the room and switches off the light otherwise. In this last example, we use some features of three different devices, but what would happen if I bought the devices a year apart from each other? I probably would forget some of their capabilities or even the presence of some of them, and therefore I will not be able to configure that kind of behaviour. The same would happen if I went into a hotel room or to an office, full of these objects, for the first time. I would take a while to be perfectly aware of their presence, of what they can do for me and to think

about their possible interactions. Moreover, each time that a new device is added to a smart space, we have one more object in our network and several new capabilities that can be combined with the existing ones.

In the worst case, a smart space is a fully connected network of smart objects where the number of possible simple interaction between two devices (c) grows quadratically with the number of objects (n), as per the formula 1.1.

$$c = \frac{n(n-1)}{2} \quad (1.1)$$

If we consider the number of capabilities that each object brings to a smart space, then the number of all possible combinations of capabilities of any length and without repetitions is

$$\sum_{k=1}^n \binom{n}{k} = 2^n - 1 \quad (1.2)$$

where n is the quantity of capabilities of a smart space. It indicates an exponential complexity of $O(2^n)$. Considering the possibility of creating longer sequences, repeating capabilities, and that the order does matter, the number of possible interactions grows further. Therefore, the potential and the complexity of our space drastically increase when we put a new object into the environment, limiting our ability to control and personalize it.

One goal of this work is to help a user to **personalize** a smart space. This goal can be achieved in several ways. Using user's preferences is possible to filter, sort and finally recommend groups of capabilities or of interactions to him. When a user enters for the first time a smart space, his previous configured interactions may be analysed and **adapted** to work with the objects available in the new environment. Users can also share their interactions created for a given space. Another form of adaptation is modifying shared interactions considering user's preferences. Adaptation can be reached easily passing through a **generalization** process where interactions built to work with specific devices are described by generic objects.

Smart spaces have an important user-centered functional component: the user is in the center of the personalization goal. The proposed approach assumes that the user has to be empowered to control the smart space in a simple, effective way. Due to the technology penetration, it seems adequate to facilitate space management through a mobile personal device, i.e. a smartphone (or similar wearable in a future).

Similarly, the user needs a simple manner to personalize the space. How can he express an interaction between objects and how can he configure them to work in that way? The **event-condition-action paradigm** (ECA) is an easy to understand solution, quite similar to the natural language. It represents sentences like WHEN an event occurs, IF a condition is satisfied, THEN do something. In the previous smart light bulb example, we used some rules based on this paradigm. Interactions between

objects will be described with this paradigm linking together objects' capabilities. A part of the experimental contribution of this work is a rule engine, that runs on the user's smartphone, which permits a simple personalization of smart spaces. It will use the ECA paradigm and it will automatically control the interactions among the objects to reach that personalization.

Nowadays several manufacturers produce smart objects following some IoT standard or a proprietary solution. In some fields, such as multimedia, the affirmation of few standards guarantees the interoperability of many devices, but in others there are a lot of protocols that make impossible the communication between objects from different manufacturers. Discovering and configuring a recently installed device can often be done only through a specific application, while the interaction between that device and another one from a diverse producer is usually impossible. Solving the **discovery** of smart objects and the interoperability among them are other fundamental aspects of this work. The employ of gateways and controllers will ensure a good level of compatibility among devices without introducing yet another IoT protocol.

In brief, when a new object is installed within a space (after the first-time installation steps: bootstrapping, connection, authentication, registering, updating and basic configuration), the first step the user has to do is to connect it to a configuration device (smartphone, tablet, pc, etc.). We call this step discovery: a control device identifies a new object and establishes a connection with it. The new object communicates to the device its features, capabilities and other parameters. The same discovery process happens when the user enters a space where he has never been before, due to the fact that he does not know the objects available. Once the control device knows the objects, the user is able to configure them or directly control them.

During the personalization process, the user configures a set of ECA rules called scenes composed of chains of capabilities. They represent a behaviour that one or more object must execute. With scenes, the interaction changes from a manual process to an automatic one. User's device automatically takes care of orchestrating objects' capabilities ensuring that they will behave as the user decided.

In order to help the user to personalize the space, a software can gather users' preferences, objects' capabilities and previous scenes. Then it can analyse, process and finally recommend preconfigured scenes to him. The process of modifying a scene passes through the generalization phase, where physical objects involved are substituted by generic objects, creating a generic scene that works in any smart space that presents the same capabilities of the original scene. Generic scenes are adapted to work in new spaces and for other users, maintaining the goal of user support. Generalization and adaptation need a **model** able to describe objects and capabilities and that permits to analyse and process scenes.

1.2 Contributions

The expected contributions of this thesis are:

1. The definition of a workflow that permits to personalize and control a smart space using the event-condition-action paradigm
2. The design of a model able to describe any kind of smart object and its capabilities.
3. The extension of the model to describe a smart space as a composition of its smart objects.
4. The application of the model to the workflow, reinterpreting the main Object-Oriented Programming features and using them to describe the interactions between objects and the recommendation process.
5. The proposal of an architecture that implements each step of the workflow and its relations with the model.
6. The proposal, development and evaluation of novel services in a realistic space.

The first contribution is the identification of which steps a system needs to: be aware of the objects available in a space and what they can do, to describe how they should behave and finally to orchestrate their interaction. Each step will be analysed, considering all the possible options for each of them. A generic architecture of a system able to implement the workflow will be also proposed.

Smart objects are the basic blocks of a smart space. The second contribution is the proposal of an abstract model able to describe any kind of object in function of its capabilities. The model creates a taxonomy of objects according to their internal sensors and actuators. The position of an object in the taxonomy determines its capabilities and it is used in the other contributions.

In a space there are several smart objects. Applying the model to each of them, it is possible to describe the whole space and its behaviour. In this point, we will show how to use the model to specify a scene.

The model is a UML class diagram, which uses and reinterprets some Object-Oriented Programming (OOP) features. The fourth contribution is how to fuse the model and the workflow: how to describe a space and its evolution along the workflow using the model. The reinterpretation of OOP features, such as method, message, inheritance and polymorphism will be used to give a solution to this problem. Particular attention will be given to the personalization step: it is one of the main goal of the thesis. In this part, we will propose some different methods to automatically help the user. The adaptation process, the use of a case-based reasoner

and the recommendation of scenes are the three solutions proposed to help the user to personalize the space.

The fifth contribution is the mapping between the workflow and the generic architecture to create a real system. The implementation of each part of the workflow in a separate block will be explained. This real system will be used for the validation of the model.

The final contribution is a real deployment of the system through the creation of some service. In this step we will exploit a software for the user's smartphone that is able to orchestrate the interaction among objects and that provides a simple interface to compose scenes.

1.3 Document Structure

This thesis is divided into seven chapters. In order to enhance the comprehension of the document, its structure is next described.

Chapter 2 reviews the state of the art. This chapter begins with a description of smart objects seen as autonomous devices and their behaviour in collaborative way. Following the workflow (discovery-personalization-interaction), the chapter reviews the state of the art of each of its steps. Finally, smart spaces models are examined, especially those related with the personalization and with the mobile-mediated interaction.

Chapter 3 covers the operational concept and the workflow that we will follow to create our model. In this chapter, all the nomenclature used in the rest of the thesis will be introduced. Next we formalize discovery, personalization and interaction separately, emphasizing the importance of the smartphone as a mediator in each step of the workflow. The chapter ends with the discussion of the architectural requirement of a system able to control a smart space during each step of the workflow. A technology agnostic architecture will be proposed in this last part of the chapter.

Chapter 4 is dedicated to the models. At the beginning, smart objects are treated as independent entities. Their internal structure and their behaviour is analysed to create a first generic model. Applying and reinterpreting the OOP features, the model is converted into a class diagram. This UML model is focused on the object's capabilities and on the components, leaving the rest of the internal structure almost incidental. Three different smart objects are described with both models.

Chapter 5 puts together the model and the workflow with the aim of describing how the system has been implemented and how it works. The technology agnostic architecture proposed in the chapter 2 is adapted to a real system. For each step of the workflow, the blocks of the architecture necessary to perform it are analysed. Each block uses the class model and reinterprets some OOP features to perform

some operation on the model. Some implementation details are also explained (data structure, software modules and shared variables) in every section.

Chapter 6 proposes four novel services: MECCANO, PERSEO, Follow-Me and the air traffic control station. These services are built on top of our system and have been implemented and deployed in our Lab for evaluation purposes. For each of them, the experimental scenario and the service architecture are described. Then, the features explained in the previous chapter are applied to the service to describe the behaviour of its scene and to validate the model and the architecture.

Chapter 7 summarizes the research of this thesis. This chapter presents a set of conclusions and contributions of the presented work. It ends by outlining some potentially future research work.

Finally some appendices describe some complementary aspects to the main work. Appendix A describes the real implementation of the orchestrator, analysing its internal components and the equivalence with the blocks proposed in the architecture.

Appendix B explains how to create a smart object and its module starting from the class model. It also includes the description of some objects that we created for the validation: a smart door and an automatic blind.

Chapter 2

State of the Art

This chapter contains a review of literature relevant to smart objects and personalization in smart spaces. The chapter begins with a brief introduction to smart objects and spaces, some definitions and basic concepts. Next, the personalization problem in these environments is examined in detail and its fundamentals are explained. Once these aspects are analysed, we include a review of several models proposed to describe smart objects and spaces. A special attention is given to the models related with the personalization. Then, some architectures for Internet of Things and smart spaces are described, focusing on various solutions proposed to overcome two big problems in the IoT: the interoperability among devices and the discovery of new things in a network. The last section treats the human-thing interaction. Since people bring smartphones at any time and in any place, we focus on the interactions mediated by a mobile device.

2.1 Smart Spaces and Smart Objects

In 1991, Mark Weiser invented the term Ubiquitous Computing ([Weiser, 1991](#)), alluding to objects that contain a computer and that can provide information easily. However, he did not specifically refer to an object as being smart, they were smart in the sense that they acted as digital information sources. The smart object concept was introduced some years later, in 1998, by [Kallmann and Thalmann \(1998\)](#). They referred to objects that can describe their own possible interactions, in contrast with *plain objects*, that do not provide information about interactions. The authors used these objects in a simulation of a virtual world to demonstrate a general interaction scheme between artificial intelligence agents that represented human beings and objects. Because of its simulated nature, these objects are known with the term virtual smart object. The term smart object used today was coined more recently: with the spread of electronics into everyday objects, there has been

a distinction between smart virtual objects and smart physical objects (or smart devices) (Manovich, 2006). Regardless of their virtual or physical nature, smart objects maintain their ability to publish their capabilities, interact with other entities and display a smart behaviour. Smart objects are also being proposed as a key enabler for the vision of the Internet of Things (Kortuem et al., 2010). Nowadays, the terms Smart Object, Smart Device, Tangible Object, Tangible User Interface and Thing in the Internet of Things are often used interchangeably and they are used to designate a wide variety of physical entities.

From the standpoint of hardware, smart objects are equipped with sensing, processing and, sometimes, communication and interfacing capabilities. The intelligence of these objects is heterogeneous: there are objects that are considered ‘smart’ just for having a NFC tag attached, while others integrate processors and memory resources that provide them with some autonomy. In brief, a smart object refers to *‘a computationally augmented tangible object with an established purpose that is aware of its operational situations and capable of providing supplementary services without compromising its original appearance and interaction metaphor’* (Kawsar et al., 2008). Following this description, a smart object may be created either as an artefact or manufactured product or by embedding electronics into non-smart object. In the first case, there are several commercially available devices such as smartphones/tablets, wearable devices, home automation systems and gadgets. The second case is mainly represented by prototypes and homemade objects.

In the literature there are several proposed methodologies about prototyping a smart object and preparing it for communications with other objects. In (Drossos and Kameas, 2007), for example, authors describe how to convert an object into a smart object improving its hardware and providing computational and communication features. Authors propose a prototype of a smart chair and a smart lamp. Another example is (Fujinami et al., 2005), where authors prototype a smart mirror capable of detecting movements and display information to the user. From this first prototype several projects are born, such as the MagicMirror, based on the Raspberry Pi board (Michael Teeuw, 2014). Looking at commercial devices, we are still at the beginning: most of them are intended to be ‘only’ remote-controllable devices instead of autonomous smart objects. Devices such as Philips Hue (Philips, 2012) (smart bulbs) (Figure 2.1a) or Lutron’s objects (Lutron, 2014) (smart dimmers, motorized shades and smart thermostat) (Figure 2.1b) may be connected to a smartphone/tablet/pc that gives them the intelligence or simply allows the user to control or receive data from them. Others, such as Nest’s thermostats (Nest, 2011) (Figure 2.1c), besides having these features, can learn from user behaviour, coordinate other devices and work autonomously. The difference between the former and the latter lies in the system complexity, in fact, Nest’s device, apart from having the typical functions of a thermostat, is able to interact with other heterogeneous objects giving them an ‘added intelligence’ and finally controlling spaces in a holistic way.

Figure 2.1: Some commercial smart objects

The concept of smart environment evolves from the definition of ubiquitous computing: ‘*a physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network*’ (Weiser et al., 1999). Youngblood et al. (2005) defines a smart space as one that is able to acquire and apply knowledge about the environment and its inhabitants in order to improve their experience in that environment. A definition more focused on the objects that configure a space is given by Cook and Das (2005): ‘*a small world where different kinds of smart device are continuously working to make inhabitants’ lives more comfortable.*’ These definitions show spaces composed of objects that work together to improve their inhabitants’ life or at least their experience when in that environment. How the space should respond to provide these tailored customization opens the challenge of personalization in smart environments. Objects should work in a coordinated way to fulfil specific goals, decided by the inhabitants themselves or by an external system able to learn from their preferences or their behaviour. The issue of how to make a space ‘smart’ is shifting from the problem of how to coordinate sensors or other elements in the space to how to provide consistent interaction between the space and the final user (Dahl, 2008). As it was previously stated, a smart space can be described as a crowd of smart objects, which acquire full meaning when they are put in relation one to each other. Besides the personalization, the interaction problem is also an open challenge; the issue is how to make easier the coordination and customization of the available network of objects for a given user. It is a challenge at ‘physical level’ (hardware and communication protocols) and at ‘decision making level’.

Among the existing projects focused on smart spaces, typical goals are, for example smart home functions, such as reducing energy consumption, providing help to the elderly, increasing usable space in a room, giving home automation control

Figure 2.2: Some smart home projects

to the dwellers, etc. In this scope, [House_n](#) (1999) was an early project of MIT Department of Architecture Research Consortium which explored how new technologies can make possible dynamic, evolving places that respond to the complexities of life. The project provided a particular attention to health and energy. [MavHome](#) (2001) was a multidisciplinary research project at Washington State University and the University of Texas at Arlington, focused on minimizing the cost of maintaining the home and maximizing the comfort of its inhabitants. [CityHome](#) (2014) from MIT Media Lab tries to maximize the usable space in an urban apartment. All these projects use smart objects (often specifically prototyped for the space) and interactions among them to achieve their goals.

Generally, people spend a good amount of time in their home performing everyday activities, so there is great interest in smart home, but there are also other smart spaces with active research projects: greenhouses, factories, buildings and cities are some of them. In this work, we will consider a smart environment as an ecosystem composed of smart objects, an entity that stores objects' information and users' configurations, and a smartphone application able to interpret one or more configurations and to orchestrate the interactions among objects, to achieve those configurations.

2.2 Personalization in Smart Spaces

Personalization is defined by [Blom](#) (2000) as a process of changing the functionality, interface information content, or distinctiveness of a system to increase its personal relevance to an individual. With the rapid development of the World Wide Web in the 1990s, personalization evolved with the aim of helping people to find the right information timely and with the right level of detail. In order to find a solution to this problem, many systems have been developed with the ability to adapt their behaviour to the goals, tasks, interests and other features of individual users or groups of users. Modern personalization systems create a user model, based

on her preferences, and apply it to select and filter information for that specific user ([Bowen and Filippini-Fantoni, 2004](#)). Blom also differentiates system-initiated and user-initiated personalization systems. The former often being described as customization or adaptation, this system needs some previous users' information to produce a different result from the default one, while the latter can work in real time without knowing any previous data. The author also point out that the two systems should be seen as two dimension of the same concept as often both the system and the user participate in the process. In ([Fan and Poole, 2006](#)), authors propose a classification scheme for personalization defining the resource to personalize, the target user and who does the personalization. In the latter point they differentiate between explicit personalization (done by the user) and an implicit personalization (done by an automatic system). Driven by social networks, in the last two years, it has also become popular the so-called predictive personalization: a kind of personalization based on a statistical analysis of aggregated user models.

In smart environments, personalization consists in adapting a space to its dwellers. It can be done by the dwellers themselves, who configure the space (explicit personalization) or directly from the environment that configure itself according to some dwellers' preferences (implicit personalization or customization) or according to previous dwellers' preferences (predictive or hybrid personalization). Since the difference between implicit and predictive personalization is linked to the source of preferences (a specific user or a crowd of users), predictive personalization is usually considered as a kind of implicit personalization and the user's preferences are included in the crowd's ones.

Explicit personalization is the process in which users directly configure and/or interact with the environment. Considering the spread of smartphones, many proposals opt for them as mediators between users and objects. It is the case of MEC-CANO ([Bernardos et al., 2013](#)), where the smartphone is enabled to discover objects in a smart space. It acts as a mediator during the configuration phase, when the user builds behaviours by combining objects' capabilities using the ECA paradigm. Finally, it orchestrates those behaviours by controlling interactions between objects. Behaviours may be shared with other users by the objects themselves, by a central server or using a cloud service. The proposed architecture also includes the possibility of using a recommender to suggest a collection of possible space behaviours to a user basing on his preferences and on the objects available in the space. A similar solution is implemented by Apple in its HomeKit ([Apple Inc., 2014](#)). Here the user can configure simple behaviours using the features published by objects using a proprietary protocol. It also permits the user to interact directly with objects, via voice control or using the device's touch screen. Google is also developing a similar framework based on Android called Android Things ([Google, 2016](#)).

Implicit personalization tries to do the same, but without direct user intervention. In (Augusto and Nugent, 2006) some different solutions to employ artificial intelligence methods in smart homes are described. The book focuses on the main techniques of artificial intelligence (case-based reasoning, neural networks and agents) and use smart homes as domain example for their applications. All these techniques are based on a centralized system that receives data from sensors, processes them and controls some actuators. Using artificial intelligence, a space may learn users' behaviour and propose them some useful configuration or it can automatically actuate to prevent accidents, for example. Other systems, mainly inspired by web services, use statistical information about dwellers to achieve the same goal. In (Musto et al., 2009) this technique is used in a museum, employing user's preferences stored in their smartphone and context information (location, noise level, brightness, etc.). The recommender creates a user specific guide service. In this case there is not a direct actuation, but this method is an alternative to artificial intelligence. These proposals are based on a central unit, others interpret the components of a space as separate entities in a distributed system, without a central entity that orchestrates them. Here a smart environment is a set of objects that can collect data and interact among them or with users. In (Parra et al., 2009), authors propose a P2P-like scheme where devices and services represent peers and act autonomously. Peers can interact directly with each other, in this manner they avoid to use a central mediator improving scalability and solving the discovery problem.

2.3 Smart Spaces Modelling

The first projects of smart spaces aimed at providing specific features such as environmental control, automation, security or multimedia. In the beginning there were different systems able to control each feature, but they were rarely integrated together. The need to integrate different systems into one global control resulted to be challenging, so new design processes started to be designed to implement global control systems for smart spaces. The core of these design processes is often a model that describes a space and the entities within. Models are designed following project specifications and observations of the real world and they are used to address the implementation of a control system. Models use different approaches according to what the purpose of the system is or to simplify some implementation aspects.

A generic approach is presented in (Kortuem et al., 2010). The aim of the authors is to introduce a hierarchy of architectures to support the increasingly complex applications, not to describe a specific smart space. In this work, objects are described in terms of three properties: awareness, representation and interaction and classified into activity-aware, policy-aware and process-aware smart objects, using the three

properties. Awareness is the ability to sense, interpret and react to events and human activities. Representation refers to an object's application and programming model. Interaction is the ability to communicate with the user. According to the level of these properties an object can be put in one of the three groups and treated accordingly. This proposal is more centred on the objects than on the spaces.

Several models follow the vision of a smart environment described using the objects that compose it. In (Fernández-Montes et al., 2014), for example, authors propose a model focused on the objects' capabilities. They briefly describe the lifecycle of an automation process in smart environment (perception, reasoning and acting). Then they propose a UML class diagram (called device abstraction) to describe sensors and actuators. This class diagram describes a set of standards and conventions for controlling, configuring and accessing the data gathered from all kinds of devices related to the smart environment. The integration of sensors and actuators is provided by following the device abstraction model to implement the system. Authors model sensors and actuators using UML interfaces, so they describe only objects' capabilities (sensing, acting and communication), but they do not model the objects' internal behaviour. This is an example of how to model a smart space using the object-oriented programming features to describe interactions and control of sensors and actuators. In (Goumopoulos and Kameas, 2009b), authors directly model smart object's internal components. An object (called Artifact) is defined in terms of properties, functional schema, states, transformations, plugs, synapses and constraints. These terms are used to describe the internal architecture of an object and how it can communicate with others. Properties are the capabilities that an object publishes to the external space. Functional schema is a set of functions that implements the properties. States are the values of the functions at a given time. Transformations are transitions between states. Plugs are connections with other objects. Synapses are associations between two compatible plugs. Constraints are restrictions over the properties. This model is used in (Goumopoulos and Kameas, 2009a) as a component of an intelligent environment that supports domestic tasks. A similar approach is proposed in (Strohbach et al., 2004), where authors define an architecture of an artefact composed of sensors, actuators, perception, inference and knowledge base. Sensors produce measurements which are converted by the perception component into data with meaning for the knowledge base and for the application domain. Knowledge base represents the situation of the artefact in the world and a part of its information is shared with other devices. Finally, inference obtains data from shared knowledge and knowledge base to generate the actions that will control the actuators. Cooperation of artefacts is tested in a real world situation in which artefacts augment chemical containers and detect potentially hazardous situations concerning their storage.

Parra et al. (2009) model a smart object as an entity composed of a Business Logic based on ECA rules that represents the computational core of the object, a Device

API that enables the communication between core and hardware, the Exposed Actions that the object publishes and the Required Inputs it needs if it is an actuator. A smart object composed of these elements is considered as a peer and the smart space is modelled as a peer-to-peer network using the DPWS protocol.

Apple’s HomeKit ([Apple Inc., 2014](#)) proposes a simple model to divide a space into different smart spaces, identify objects and control them. This framework is oriented to a smartphone-object communication, where the smartphone acts as a master and the objects as slaves. Objects (called Accessories) share their capabilities (called Services). The smartphone discovers the objects and organizes them according to their location to create different smart spaces (homes, rooms or zones). Finally, the user can manually control an object by interacting with its shared services or he can mashup the services in configurations called Scenes that can work without the user intervention. HomeKit is a framework, not a model, and thus its aim is to facilitate the development and integration of new devices and services. Hence its object description, only based on services, regardless of their internal structure and behaviour, is very simple.

Another completely different approach is proposed in ([Seiger et al., 2014](#)). It is based on modelling the processes that take place in a smart environment instead of the objects that compose it. Authors name the model Process meta-model. Processes are described using the Meta-Object Facility standard (MOF) that extends the UML language. With MOF authors are able to describe simple processes (called Steps) using the UML language, while complex processes can be modelled using UML, when it is possible, or with non object-oriented models such as Petri nets. This solution is totally object agnostic, it only models the events that occur in a process.

2.4 Architectures for Smart Spaces

The novel paradigm named the *Internet of Things* was first introduced by Kevin Ashton during a presentation in 1999 ([Ashton, 2009](#)). It designates a network of devices, vehicles, buildings and other items where they are able to collect and exchange data. The IoT represents the connection between the physical world and computer-based systems. Today the term is used as a catchphrase to denote not only a global network of items, but also small local networks of objects not connected to the Internet. It is the case of Wireless Sensor Networks (WSNs) and Mobile Ad-hoc networks (MANETs) that can be considered part of the IoT ([Bellavista et al., 2013](#)).

Smart spaces are compositions of smart objects interconnected among them that perform tasks mainly related to a specific environment, but, if necessary, could also

exchange data with remote devices, consequently they are a part of the IoT. Therefore, architectures for the IoT also apply to smart spaces.

Architectures include physical components and their organization and configuration, services delivered and network protocols. The IoT-Architecture Consortium drafted a series of best practices, guidelines and a starting point to generate specific IoT architectures, which are collected in IoT Architecture Reference Model (ARM) ([IoT-Architecture Consortium, 2013](#)). The Consortium uses a metaphoric representation to give a global vision of an IoT architecture. They arrange nodes and protocols, middleware and applications in a tree where the firsts are the roots, the second is the trunk and the third are the leaves. They also provide a set of deliverables that deal with specific issues of IoT architectures, such as interoperability, availability, performance, security, etc.

[Bandyopadhyay and Sen \(2011\)](#) separate the architecture in four layers: edge, access gateway, middleware and application layer. Edge layer groups the hardware and the general items' capabilities of identification, storage, collection, processing, communication, control and actuation. Access gateway layer defines the routing protocols, subscription and publication methods, and cross platform communications. Middleware layer encompasses the mechanisms of communication between hardware and applications. Finally, application layer gathers user interfaces and services provided by the underlying network. Each layer uses its own hardware and software technologies. Also [Misra et al. \(2015\)](#) see a general architecture divided into four levels: physical/virtual space, sensor/network as a service, "Big-Little" data management and on top analytics and decision making. Comparing the two works, the four layer division almost coincides. A similar division was proposed by [Wang et al. \(2015\)](#) using only three layers: nodes and their operating systems, middleware and data storage via Resource Description Framework (RDF). They group physical devices with their operating systems and network protocols in the lowest level. In middleware, they include protocols that allow communications between objects (routing) and with applications. In data storage, they propose to use the RDF specifications ([Klyne and Carroll, 2004](#)) to provide a standard communication between applications.

The layer division is quite similar in the three surveys: Bandyopadhyay and Sen and Misra et al. consider gateways as a different hardware from nodes, which main task is the routing, and put them on a separate layer; Wang et al. include gateways' hardware among the nodes and assume routing protocols as part of the middleware. All the three surveys and the IoT-A ARM agree that middleware is the critical part of a IoT architecture. [Bernstein \(1996\)](#) stated: *'For many new applications, middleware components are becoming more important than the underlying OS and networking services on which the applications formerly depended'*. Middleware should solve problems such as interoperability across heterogeneous devices, context awareness, device discovery, management of resource-constrained embedded devices, scalability,

management of large volumes of data, security and privacy. In this thesis we will use networks of heterogeneous devices, so we focus next subsections on interoperability and discovery. The term middleware is also often used to denote platforms and frameworks.

Platforms for IoT generally define a group of network protocols that they can use, a set of APIs to build applications using the platform and some software components that perform a particular task such as brokers, drivers, services and a core application that coordinates all the elements.

Smart-M3 ([Honkola et al., 2010](#)), for example, is an open source platform thought to enable smart environments using a semantic web infrastructure. It uses the RDF to describe devices and their capabilities and TCP/IP, HTTP and Bluetooth as transport mechanisms between software and devices.

Some companies offer a IoT platform and design, for each customer, an ad-hoc infrastructure to connect devices. It is the case of Intel IoT Platform ([Intel, 2015](#)). [Qivicon \(2011\)](#) is an alliance of companies founded by Deutsche Telekom. It has its own commercial platform designed for home automation. It is composed of a control unit (a gateway called Home Base) that enables a bidirectional communication between various devices installed in a smart home and a cloud service. Users can control their devices remotely using a smartphone/tablet application or a web page. Home Base gateway is an hardware device that needs a connection to a modem/router to reach the Internet, but a future integration of this gateway in a modem could enable Internet Service Providers to offer new kind of services to their customers.

2.4.1 Interoperability

The interoperability problem arises when devices from different manufacturers or using different protocols try to work together. [Tschofenig et al. \(2015\)](#) identifies four basic communication models: Device-to-Device, Device-to-Cloud, Device-to-Gateway and Back-End Data Sharing.

The first one concerns the direct communication between two devices (that use the same physical and data link protocol), rather than through an intermediary. This is the worst case for interoperability: only device manufacturers can solve it using open standards or developing additional features that enable compatibility with other manufacturers' devices. These incompatibilities limit user choice to devices within a particular protocol family. For home automation, KNX standard ([KNX Association and others, 2004](#)) ensures a direct communication among objects connected to the same physical layer, regardless of manufacturer, while it uses gateways if they do not use the same physical channel. The problem of this standard is that it has inherited its feature from old standards such as BatiBUS, EIB and EHS,

so it maintains communications over twisted pair wires, powerlines, radio (KNX-RF), infrared and Ethernet cables, but it does not natively support new physical layer standard more oriented to the IoT such as IEEE 802.15.4 or Bluetooth. Moreover, it is only oriented to home automation.

In Device-to-Cloud communication model, the objects connect directly to an Internet cloud service. This time, the devices must be able to communicate, through an IP network, directly with a remote server on the Internet. Several vendors use this model to enable the user to obtain remote access to his objects via smartphone or Web interface. In this case, interoperability challenges can arise when attempting to integrate devices made by different manufacturers, if the objects use a proprietary protocol to communicate with the server or if the cloud service does not provide a manner to access its data.

Device-to-Gateway model combines the two previous ones. It adds a local gateway that splits communication among devices and communication with cloud services. Interchange of data between the two domains must pass through the gateway. In this manner, local networks of devices can use non-IPv4 protocols, but they maintain their capability to communicate with a remote server on the Internet. Moreover, extending low layer protocol features on the gateway, it is possible to achieve the communication with several devices from different manufacturers. The *Smart-Things* hub ([Samsung, 2014](#)) is an example of gateway able to communicate with both Z-Wave and ZigBee devices and to provide them the ability to connect to a cloud service. With this gateway, two devices that use a different standard are able to communicate (indirectly) between them and the user can remotely control them. The last model enables an interoperability at application level. It is an extension of the Device-to-Cloud model where cloud services can exchange data using standard protocols. In this case, vendors must provide cloud applications programmer interfaces to grant access to data stored on their servers. The extension of this model is the Web of Things (WoT). The idea behind WoT is to reuse existing Web standards such as HTTP, REST, JSON, Websockets, OAuth, etc, and to apply them to the IoT. Using these standards, it should be easier for vendors provide a good level of interoperability, at least at application level. The Constrained Application Protocol (CoAP) ([Bormann et al., 2015](#)) proposes to give to the devices the ability to communicate directly with web services through RESTful interfaces. CoAP is an application layer protocol that works with HTTP, UDP and IP. Considering the Internet protocol layer, IPv6 ([Deering and Hinden, 1998](#)) was born to connect to the same global network all types of devices. In order to solve problems related to the physical layer and to allow to the smallest devices to connect to the Internet, a reduced version of IPv6, called 6LoWPAN ([Montenegro et al., 2015](#)), has been proposed. Despite this, adoption of IPv6 is still not widespread and device manufacturers prefer to maintain their link layer protocol and IPv4 for Internet connections. Due to this fact, the use of gateways is currently the best solution

for interoperability and it is the most commonly used among objects that use the smartphone as a gateway (for example personal fitness tracker or e-Health devices).

2.4.2 Discovery

Interoperability ensures that objects and services are able to exchange data among them, but before establishing a communication channel, each entity needs to know with whom it can communicate, what kind of data can understand the receiver and what it can do with that data. The process by which an object becomes aware of this information is called discovery. The process is often bidirectional: a device learns what the rest of the network can do and the network learns what the new object can do.

Among standards, several protocols include discovery. UPnP ([Forum UPnP™, 2008](#)) is one of the most popular. It was designed to permit devices to seamlessly discover each other's presence and the services provided by them, and finally create a network able to work without user intervention (following the zero-configuration networking principles). It is based on standards such as TCP/IP, HTTP, XML and SOAP. UPnP clearly separates the discovery into a *step 0*, where devices get a network address, and *step 1* that is the real discovery. Step 0 is performed using DHCP or other similar protocols. In some kind of networks composed of embedded devices, such as WSNs, MANETs or PANs, this step includes the 'physical discovery' of the object (device's neighbours, some special nodes or the entire network become aware of it) and its inclusion in the network and address assignment, if necessary. IP networks often employ DHCP, while networks based on other standards have their own protocols that usually also include route discovery (AODV, DSR, ZigBee, etc.). Step 1 is more oriented to discover features and services that a device provides. In UPnP, step 1 is carried out using Simple Service Discovery Protocol (SSDP). The latter has been incorporated into UPnP and it is currently described in UPnP standards documents. Devices that use UPnP announce their presence to the rest of the network sending a SSDP message to a multicast address using UDP and publishing their universally unique identifier (UUID) and their capabilities. Some parts of UPnP are used by other standards. Digital Living Network Alliance (DLNA) ([DLNA Organization, 2014](#)), for example, uses its discovery, resource sharing and device control method. DLNA is oriented to multimedia content sharing, so it cannot work with devices intended for other purposes. DLNA's direct competitor is Apple's AirPlay ([Apple Inc., 2011](#)). It is a proprietary protocol also oriented to multimedia, and it uses Bonjour ([Apple Inc., 2002](#)) for discovery. Bonjour only works in local networks, and it is used to discover devices in a lot of Apple's applications, such as the aforementioned AirPlay, AirPrint, HomeKit, etc. Bonjour is built on top of multicast DNS (mDNS) ([Cheshire and Krochmal, 2015b](#)), a standard widely used in the IoT. The protocols mentioned are used to discover devices and their services.

Other protocols are used only for services. It is the case of Web Services Dynamic Discovery (WS-Discovery) (OASIS, 2009) and DNS-based service discovery (DNS-SD) (Cheshire and Krochmal, 2015a) that are used to discover services on a local network regardless of the devices that provide them.

All the cited protocols follow the Zero-configuration networking (zeroconf) paradigm: a set of technologies that automatically configure a network without requiring manual user intervention. Kim et al. (2012) include these protocols in automatic discovery: a process where, when a smart object is turned on, it automatically announces itself to the other devices in the same network, publishing its capabilities. Authors also define manual and semi-automatic discovery.

In manual discovery, user must manually add all information about the device to the gateway. This process is often used when the device has limited hardware, and therefore it cannot execute a discovery protocol or it uses a communication technology that does not have any discovery protocol implemented on it, such as X10.

In semi-automatic discovery the user manually begins the discovery process, but it is automatically completed by the involved components. The process starts when the user presses an hardware button on the device or on the gateway, when he inputs the hardware address or a code related with the object into an application or a web form, or when he reads a barcode or a NFC tag with his smartphone. In case the user presses a button on the new device, it announces itself to the rest of the network, beginning a discovery protocol. If the user presses a gateway's button, it is the gateway that initiates the process to recognize all the objects available, the old ones and the new one. This method is used, for example, by the Philips Hue gateway (Figure 2.3a) to discover its lightbulbs. If the user inputs the object's network address or a code associated to it into an application or web service, a gateway connected to the application begins a unicast discovery towards the network address related to the input code. It is used, for example, to add some devices to the Apple's Homekit (Figure 2.3b) (it is also possible to use the camera to scan the code instead of adding it manually). Finally, if the user interacts with the object reading a NFC tag or a QR code, the latter is converted into a URL by a gateway. At the URL is located a bundle that contains object's information. The following download and installation of the bundle are automatically performed by the smartphone. If the new device uses Bluetooth (at least v2.1) the code conversion to URL may not be necessary: Bluetooth Secure Simple Pairing (SSP) implements a direct pairing mechanism that works with both NFC and QR code and automatically connect the smartphone to the object. Also for WiFi exists a similar feature: if the new device is the smartphone itself, it can enter a protected WiFi network reading a NFC tag or a QR code, and then initiate a discovery protocol built on top of IP (Figure 2.3c). This kind of discovery is used when the device has limited hardware and it cannot implement a discovery protocol, or when the initial interaction is performed with a

Figure 2.3: Three kinds of semi-automatic discovery used in commercial devices

smartphone and the new object does not have the same communication technologies that the mobile has (for example ZigBee).

Commercial products generally use automatic or semi-automatic discovery due to their ease of use.

Also recent research works are focused on automatic and semi-automatic discovery. Most of them use one of the aforementioned standard protocols. [Klauck and Kirsche \(2012\)](#) for example, integrate Bonjour in Contiki operating system for embedded devices, creating a uBonjour version. [Parra et al. \(2009\)](#) use Devices Profile for Web Services (DPWS) ([Driscoll and Mensch, 2009](#)), which includes WS-Discovery, to create some smart objects able to communicate among them with that standard and considering them peer node and the smart home environment as a peer-to-peer network. [Kawamoto et al. \(2007\)](#) propose a gateway able to connect ZigBee devices to IP objects using DLNA to enable plug-and-play, and its SSDP for discovery.

2.5 Mobile-Mediated Interaction in Smart Spaces

The user experience in a smart space is linked to the ease of configuration and especially to the natural interaction with the environment ([Thompson, 2005](#)). Nowadays there are 7.22 billion mobile devices in the world ([Boren, 2015](#)), a quarter of them are smartphones and its share will double by 2018 ([eMarketer, 2014](#)). Smartphone users interact with their devices 10-200 times a day on average, with a total time of 30-500 minutes per day ([Falaki et al., 2010](#)). The increasing number of sensors, communication capabilities, compatible devices and available applications push the users to use their smartphones increasingly in every thing they do. In this scenario,

using the smartphone to enable the user to interact with the environment seems to be a good idea.

The potentiality of mobile devices to interact with the environment has been considered both in literature and in commercial applications from some years now. For example (Beigl, 1999) explored how to use traditional mobile messaging to configure an scenario in which some objects were capable of sending SMS with simple commands to facilitate their remote control and operation from a mobile device and (Want et al., 1999) shows how to benefit from inexpensively RFID tagged objects through portable computers.

From the commercial point of view, there are a number of applications designed to act as control centres for smart homes. In the previous sections we have cited Nest learning thermostat, a device that, apart from being a thermostat, is able to connect various devices and to provide a remote control of all of them to the user through a smartphone application. Many other devices have a dedicated application-based interfaces to support a set of limited actions on them, making possible for the user to switch the devices on and off or configure a set of alerts or other simple actions. It is the case of Philips Hue and LIFX for smart bulbs, Lutron and ZWave devices for home automation, iZone and iBaby for camera-based security, Lego Mindstorms EV3 and Parrot drones for gaming, Runtastic and Fitbit for health and fitness, etc. Almost all manufacturers have their application to control their set of objects. Nest and Google with Weave and Apple with HomeKit try to provide a common protocol to control several devices with only one application.

A little different is the case of multimedia. Here Digital Living Network Alliance (DLNA) / Universal Plug and Play (UPnP) dominates the market. It enables the user to control all compatible objects through other devices (smartphone, tablet, computer, remote, etc.) or applications. Almost all smart TV nowadays are DLNA compatible. The main competitor is Apple with AirPlay but its spread is limited to Apple ecosystem and few other devices.

All these solutions enable the user to configure the objects and to interact with them but always through a smartphone app using the touch screen. This only converts the smartphone into nothing but a remote control panel, a solution quite far from a natural interaction with the environment.

Apart from these commercial proposals, a number of research works have addressed the development of system architectures to make possible the interaction between mobile devices and objects. Some works focus on taking advantage of mobile sensors to design physical interaction methods between the device and the object, e.g. using touching, scanning or pointing solutions (Rukzio et al., 2007). For example, (Pohjanheimo et al., 2005) and (Lampe et al., 2006) propose two systems that implement the ‘TouchMe’ paradigm, a concept that allows accessing virtual services by touching the involved object, typically with a Near Field Communication

reader. In the first work, authors implement different services such as picture synchronization by touching a computer, phone call by touching a business card or check book availability in a library by touching the book. In the second work, authors enhance the gaming experience of a board game by touching the physical characters on the board. In (Hardy and Rukzio, 2008), authors propose an architecture that also uses NFC to interact with displays, in order to overcome the screen limitations of mobile devices and establish a bidirectional channel between displays and devices through actions such as ‘select & pick’ or ‘select & drop’. The EMI2lets multi-agent architecture (De Ipiña et al., 2006) is designed to control objects through basic actions, from a mobile device or a web service. The architecture is demonstrated with objects tagged with visual markers.

In (Beigl, 1999), a stand-alone ‘remote control’ is used to get the control information from other devices in order to allow operational interaction through a simple user interface. In a similar way, (Broll et al., 2009) proposes to use a laser-equipped device to retrieve a set of control commands from an object: when the object detects the laser beam, it sends the control description to the master device by using infrared. Some other proposals intend to take advantage of gesture-based interaction. In (Pering et al., 2007), for example, authors propose Gestures-Connect, a system that uses both NFC and acceleration-based gesture recognition to make selection and action on an object: a user scans an NFC tag placed on the target object, then a Bluetooth connection between the smartphone and the object is established; then the user can interact with the object using the smartphone screen or through gesture detected by an accelerometer attached to the mobile.

Many of the described proposals are simply designed to facilitate the access to information or services, not to control the objects themselves. In those cases that include some control mechanism, control options are to manage a single object. Some solutions provide a more advanced control level enable the user to mashup capabilities and services from different components. From the commercial point of view, IFTTT (IFTTT, 2011) is one of the most used applications with about 20 million recipes available for people (Lunden, 2015). IFTTT is the abbreviation of ‘If This Then That’. It was initially thought to allow users to create chains of simple conditional statements (the recipes) using web services such as Gmail, Facebook, Instagram, etc. An example of a recipe might consist of: when the user posts something on Facebook, then post the same content on other social networks. This application is based on web services, but as many smart objects are also providing web services, IFTTT began to integrate them, allowing users to create recipes of objects. Devices such as Nest thermostat, LIFX, Luton and WeMo in home automation field, and Fitbit, Withings and Nike+ between fitness and wearable are compatible with IFTTT. The main limitation of this system is the availability of capabilities implemented as web services by the manufacturer: often a small subset of a smart object’s capabilities is released and only the official mobile application

provides all of them. Apple HomeKit works in a similar way, but it was designed especially for smart objects and home automation. It is a framework directly integrated in the iOS operating system that allow the user to easily discover the objects available within a space, control them directly one by one or mashup their capabilities in simple scenes. It was thought to integrate the communication with the object in the operating system, scene creation is a secondary feature, nowadays rarely used. Moreover HomeKit is a closed framework, so the number of compatible objects on the market is limited. Google proposed in early 2015 the project Brillo, a modified version of Android for smart objects. A stable version of that operating system is currently available with the name of Android Things. It should provide the same features of HomeKit, but using an open system. The communication protocol will be Weave, designed by Google and Nest and already implemented in Nest thermostat and several other smart objects.

The application developed for our research work MECCANO ([Bernardos et al., 2013](#)) also allows users to discover objects and create chain of capabilities following the ECA paradigm. Our system was initially thought to control sensors and actuators of the smartphone and create behaviours such as put in silent mode when at work or switch on the WiFi when is charging. It also had a social component: users can share their compositions. Then we extended the system to external objects and we included a recommender. Now this application is able to interact with a lot of home-made and commercial objects and to recommend shared behaviour to a user according to his preferences. Smartphone's capabilities and objects' capabilities can be mixed to create complex behaviours such as switch on the television and turn off the lights when I sit on the sofa and it is later than 8:00 PM or pass a call to an external speaker if I am at work and who is calling is in my work contact list.

2.6 Summary

In this chapter, we have reviewed the state of the art of several aspects related with smart objects and smart spaces. The history of smart devices is strongly bound to the evolution of the electronics industry, the increase of the integration level has permitted to create new objects with advanced features. Adding new features means finding new challenges to tackle.

Nowadays, most of the commercial devices are intended to be “gadgets”, controllable from a smartphone, tablet or computer. They have been created thinking in a one-to-one relationship, where only the object and the controller exist. The integration with the rest of objects in the same space is barely considered. In home automation, bridges allow to integrate several devices to work together in a smart space, but often their role is only to provide connectivity. Big technology companies have their own frameworks for the integration, but they produce hardly any smart object and the

frameworks are still young and only designed for home automation. A common model for both smart device design and interfacing with the controller is necessary to establish clear rules about configuration, interaction and mode of operation of any kind of smart object. The models reviewed have been created a posteriori, to describe specific prototypes and their behaviour, not to define some guidelines to make an object or to create common interfaces to control them. On the other hand, they provide a good starting point to draw a new one able to cover all the aspects mentioned above.

Personalization is one of the big challenges of smart spaces. The current object-controller interaction model is focused on customization: the controller explicitly configures the object according to the user's preferences. Personalization means that with the preferences stored in a device (the controller), the whole smart space automatically sets up itself by interpreting these preferences. This process should be space-independent and auto-adaptable: it should work in any space and it should auto-reconfigure itself when new objects enter or leave a space. The next step will be to learn the user's preferences without an explicit definition from the user.

In this aspect, web pages are currently the best example of personalization: they try to adapt the graphical components and content of the page to the user's preferences, which are obtained directly or indirectly. In the literature, the personalization process is classified according to several parameters (e.g. who begins the process, how the preferences are gathered, etc.). Several works have been reviewed, each of them follows a personalization paradigm. The use of some AI technique is probably the right way to analyse the preferences of a specific user and to extrapolate a space configuration from them, at least for home automation. In other kinds of environments, for example where there are a lot of users, a statistical analysis may be better than AI. In any case, a universal solution is difficult to find. Some web services probably implements very good personalization algorithms, but they are closed and limited to the virtual world. It is the case of Google, Amazon and other big technology companies, for example. Tracking the behaviour of the user when he uses Google's or Amazon's services, the personalization algorithms are able to provide other services and/or advertisements for that particular user. A good part of the technology companies' earnings depend on the efficiency of the personalization that they are able to provide and their algorithms are closed and secret.

Coming back to the physical world, a smart space can be interpreted as a set of smart objects, connected to create a network, that interact among them and with the orchestrator. Architectures for the IoT can be used to create that network and to define the communication protocols that objects must use to cooperate. Several alternatives and standards have been proposed for this purpose, but there is not a predominant one. Complex solutions solve many problems of these kind of networks, but they require more complex hardware, that is often unavailable on the devices. On the other hand, simple protocols does not provide a fully functional solution and

they require a specific adaptation for some issues that may happen.

Heterogeneous networks have various problems due the different nature of the nodes. Discovery and interoperability are two of them, that should be solved to provide a good user experience.

Discovery is the process that permits to include a new object in a network and to know its capabilities. In the literature, various alternatives have been proposed. Most of them derive from IP-based network protocols (e.g DHCP, DNS, etc.) and they have been adapted to the IoT. Moreover, depending on the complexity of the devices, they may require the user intervention. We distinguish three kinds of discovery: automatic, semi-automatic and manual. Each of them has various possibilities, proposed in both research works and commercial applications. For automatic discovery, the UPnP is nowadays perhaps the best option. It is a well known standard, available in many devices, that have been extended for specific applications, such as the multimedia with the DLNA. Manual and semi-automatic discovery adopt ad-hoc solutions to overcome the issue of announce a new device to the rest of the network. The interoperability is the other big problem. Objects that use different communication technologies should be able to exchange data. If creating a homogeneous network is not a possibility, the only way to ensure the interoperability is using one or more gateways. It may be complemented with the adaptation of the network protocols (e.g. address translation) to provide a transparent communication between two different objects that use two different technologies (e.g. 6LoWPAN and IPv6).

Finally, in the last section, we reviewed the research work about the mobile-mediated interactions. The heavy use of the smartphone to perform everyday tasks suggests that controlling smart objects through a mobile device may be a good solution. Since the beginning of the mobile phone era, the use of the mobile devices to control other things have been explored. First interactions used SMS, later, with the integration of new sensors and the development of the technology, new kind of interactions are born. Almost any kind of sensor available on a smartphone has been used: camera, microphone, accelerometers and inertial sensors, GPS, NFC. Regardless of the kind of interaction, all theses research works demonstrate that it is possible to use a smartphone to interact with smart objects.

Chapter 3

Operational Concepts and Workflow

This chapter analyses each step of the interaction user-environment. A very important point of the thesis is the kind of the interaction. The *Mobile-Instrumented Interaction* used in the title of this work means that all the interactions are controlled by the user's smartphone, but the role of the smartphone is transparent to the user. The user interacts with the smart objects that compose the environment in a natural way, the smartphone only “controls” the process, adding additional information or coordinating some behaviours.

The content of this chapter will be the following. First, it will discuss the operational concepts that can be found during the interaction. The discussion will be approached with a hypothetical scenario where a user interacts with various smart objects in some different spaces. The first section will also introduce some nomenclature that will be used throughout the rest of the thesis. This will be followed by the identification of a general interaction workflow suitable for any smart space. The workflow will draw the guidelines to design a smart object model and to define an architecture for smart spaces in the following two chapters. Finally, a general technology agnostic scheme that identifies the main components of a user-smartphone-environment system will be proposed.

3.0.1 A Scenario

Let us consider the following scenario. *Bob is a businessman, fond of technology. Every day he goes to work. In his office he has a desk with a personal computer, a reading lamp, a smartphone dock and a digital frame with a photo of his family. In the office there are also a HVAC system and automatic blinds. When Bob arrives at the office, he is used to turn on the computer, set the temperature at 23 °C, raise the blind, connect his mobile phone to the dock and play some classical background music from a playlist on his smartphone. In the afternoon, when the sunlight reaches*

his desk, he puts the blind down and turn his reading lamp on. Before leaving the office he turns off the HVAC system, the computer, the music and the light. One day Bob realizes that all the objects in his office are smart objects (objects able to work in an autonomous way), so he plans to automate all behaviours he makes every day. He opens an application on his smartphone, which automatically discovers the surrounding objects. Thus he configures: when he connects the smartphone to the dock and it is between 8:00 and 9:00, the HVAC turns on set to 23 °C, the blind raises, the computer turns on and the music starts playing; when the sunlight reaches the light sensor of the smartphone, the blind goes down and the reading lamp turns on; when he disconnects the smartphone from the dock and it is later than 19:00, the HVAC, the computer, the music and the light turn off; every day at 12:00 the digital frame replaces the photo of his family with another one from the same album on his smartphone.

This short scene gathers the key concepts we want to approach. The whole interaction relies on a smartphone application, which serves as an intermediary between the user and those objects in his vicinity. The interaction begins when the smartphone discovers the smart objects and their capabilities. Then, the user can compose behaviours using these capabilities. Finally, the smartphone application provides a real time orchestration of the objects.

In that first scenario are shown some basic feature of the system that we will discuss in this thesis. Other most advanced features, that will also be dealt with in this work, are shown in this second part of the scenario reported below. *At 20:00 Bob arrives home and he opens the application, which ‘discovers’ the home HVAC system, the Hi-Fi and a smart socket. The application then asks Bob if he wants that, when the mobile is plugged in in the living room, the HVAC turns on at 23 °C and the Hi-Fi plays a music from his playlist. During the weekend he buys a digital frame for home. He uses the smartphone to discover the new object, when it is found, the application asks Bob if he wants to show a photo from the family album stored in the smartphone and to change it every day at 12:00. The following week Bob has a business trip and he stays in a hotel. When he enters the room he takes the smartphone and discovers a smart TV, an alarm clock and the main door lock. Then the smartphone asks him if he wants that the TV plays a song from his playlist and it shows a photo of his family when he enters the room (when he opens the main door lock). Moreover, as many other guests usually set the alarm clock at 8:00, the smartphone asks Bob if he wants the alarm rings at 8:00.*

This second part shows other concepts. Behaviours will be portable to other spaces composed of similar objects. An external entity can propose behaviours based on user’s preferences or on behaviours shared by other users for that space. All the notions reported in the scenario will be analysed in detail in the next section.

3.1 Operational Concepts

Using the above mentioned scenario, this section will describe the characteristics that a system must have from the viewpoint of a user who will use it. It is important to note that we only consider ‘complex and automatic’ interactions such as those described in the scenario. Simple smartphone-object interactions, such as *user opens the official application for a smart bulb and he switches it on remotely using the app*, will not be considered. All producers of smart devices provide through their apps this kind of basic interaction, but they cannot deliver a full integration between the user and all the objects within a space, unless all the devices are from the same manufacturer.

The device, that enables the **interaction** between the user and the space, is the user’s **smartphone**. It discovers the **smart objects**, configures them and orchestrates the interactions.

In order to enable any form of interaction, first the smartphone must know which are the objects around it and what they can do. We call this process **discovery**. In the ideal situation, the user unlocks the smartphone and he sees the available objects on the screen (**automatic discovery**). In a real scenario, not all objects are able to announce themselves and, in some cases, the user should locate the physical objects and perform some interaction with them (**manual discovery**).

Apart from becoming aware of their presence, the smartphone receives what objects are capable of doing: a collection of objects’ **capabilities**. A capability is an operation, simple or complex, that an object can do and that produces a result. This result can be an **event**, a **condition** or an **action** (**ECA paradigm**). Events are generated by **sensors**: devices or components that observe the real world and report values about physical parameters.

An event is a variation of an observed parameter, regardless of its complexity and processing level: they can be for example a rise of temperature above 23 °C measured by a thermometer or the presence of more than five people moving simultaneously in a room detected by a camera and processed with image recognition. In brief, an event is a variation generated by the environment, detected by a sensor and sent as a pulse to the smartphone.

A condition is the Boolean reply to a query, on a value or state. It implies a bidirectional communication: a query sent by the smartphone to a device, and a reply sent the object and received by the mobile. The query might be sent to any kind of device or entity. For example, the smartphone can ask to a thermometer if the temperature is above 23 °C, to a bulb if it is switched off or to a weather forecast predictor if tomorrow it will rain. All replies will be in the form yes/no (it accomplishes or it does not accomplish the condition).

Actions modify the environment activating or deactivating an **actuator**. An actuator is a device or a component that directly interacts with the real world (this also implies providing information to the users). As for the sensors, the interaction between smartphone and actuators is done using pulses sent to the objects that control them. The level of complexity of an action is irrelevant, so are actions: switch on a LED, open all the windows of a building, maintain the temperature at 23 °C or show me the latest news.

The use of simple signals to communicate events, conditions and actions requires that the objects involved have been previously **configured** to work in a certain way. If an action for an air conditioner is *to maintain the temperature below 23 °C*, it will be impossible to execute it, unless the air conditioner has been previously configured with that temperature value, or using a different communication method. In order to simplify this aspect, events, conditions and actions can be accompanied by one or more **parameters**. A parameter is an additional data that complements an event, condition or action. It can be used to send or receive information from external devices. Thus, in the previous example, the smartphone will send an action of *start maintaining the temperature* with the parameter *below 23 °C*. But the air conditioner can also be preconfigured to maintain the temperature below 23 °C and the action sent by the smartphone will only switch it on. Both methods can be used, depending on the complexity of the smart object.

The ECA paradigm, with the aforementioned type of events, conditions and actions with parameters, allows composing complex behaviours, but it does not have any chance of interaction with objects that are not involved in a behaviour. Configurations such as *when the alarm sounds, show me (on my smartphone) the outside temperature measured by my weather station*, cannot be carried out, unless the weather station has its own interface (application or web service) on the smartphone. In fact, in this case the action is *show me the temperature*, the place from where retrieving data cannot be described using ECA and parameters. If the smartphone had an application able to retrieve data from the weather station (and it was able to receive and execute external commands), the action would be sent to this app instead of the weather station. Thereafter, it will be the application in charge of retrieving data from the station and show them to the user. If the device is REST compatible, a web browser or a RESTful service can replace the application. This situation could require a third party application for each object or devices able to communicate with the smartphone through mobile-compatible standard protocols such as RESTful or IoT protocols. In order to solve this problem, the smartphone and the objects share a zone of memory, located on the smartphone, where they can read and write data: a sort of blackboard. These data are called **variables** and enable the communication among events, conditions and actions. Moreover, smartphone and smart objects have a mechanism to request a variable without using an event, a condition or an action. In the previous example, the smartphone can ask

the weather station for the current temperature and the weather station will reply putting the read data in a variable. This kind of data request is not part of the ECA interaction, but the interaction uses the produced data. Variables are also useful to store historical data such as last caller id or filename of the last picture taken. Variables are usually linked to one or more object's capabilities: it is highly likely that a thermometer, able to trigger an event of temperature above 23 °C, is also able to read the current temperature. Variables are shared by objects, with capabilities and parameters, during the discovery phase.

The smartphone application looks for smart devices in its vicinity and it learns their capabilities. When it detects an object, it downloads a piece of software associated to it. This piece of software is called **module**: a sort of **driver** that describes the object's capabilities, parameters and variables, and instructs the application to use them. In the case of automatic discovery, the device itself can store and publish its module. In case of manual discovery, the user interacts with the object and the interaction initiates the download, directly from the device if it can store the module, or from an external **server** otherwise.

With the downloaded modules, the user can create one or more **scenes** that describe the behaviour of the smart space. A scene is a particular composition of several smart objects that work in a coordinate way. The objects are the 'actors' in a scene and the smartphone application is the coordinator that **orchestrates** them. Events, conditions and actions are the objects' capabilities. Scenes are configured using the ECA paradigm, an approach easy to understand and use for the user. Using this paradigm, objects trigger events, check conditions and execute actions. The composition of some events, conditions and actions generates a scene that the smart space is able to perform. The smartphone controls and coordinates all interactions. When an event is detected by a sensor it is sent to the mobile that activates a scene. If there is a condition, the smartphone sends it to the relative object which checks the condition and returns the result. Then the mobile activates one or more actuators according to the scene.

Several scenes may be grouped together by assigning a label to each of them. A group may represent a **service**: a set of ECA rules that should work collectively (sequentially or concurrently) to produce a complex behaviour that only one scene cannot provide. The labels may be also used by the user to organize them and no other purpose.

Scene, space behaviour, ECA rule and ECA composition are synonymous. The creation of a scene represents the **personalization** of a smart space. Looking at the scenario, there are two possibilities of personalizing a space. At the beginning, Bob **manually creates** his own scenes with his smartphone. When he arrives at home, the smartphone **automatically proposes (recommends)** to him some scenes using the ones he created for the office and the objects available at home. When he goes to the hotel, it recommends scenes using Bob's **preferences** (get from the

scenes stored in the smartphone) and **(similar) scenes** created by other previous hotel guests. The first case is an example of manual creation where the scene are composed without any external aid and they are stored only in the smartphone. Scenes are not modified by other entities or stored elsewhere. In the second case, a scene is ‘ported’ from the office to home: with the necessary changes, it will work in the new space (at least in a similar way). The HVAC system is identical object at home and in the office, while PC and Hi-Fi are not identical, but both have the capability to play music. In addition, the smartphone docking station and the USB cable are not the identical object, but both provide power to the smartphone and generate the same event of plug-in. The smartphone recognizes the capabilities involved and it replaces office’s objects with the home’s ones, maintaining the same events, conditions and actions. **Adapted scenes** are always proposed to the user, who can always decide whether to accept and enable it or not. If in the new space there is not a compatible object, the scene may be proposed in a reduced form or even not to be proposed. An example of a similar, but not identical scene is *play the music and show a photo on the smart TV when Bob opens the main door lock* proposed to Bob when he is in the hotel room. The scene is a mix of the two scenes recommended when Bob is at home. Smart TV at the hotel is able to play video, audio and image, so music and photos will be played there. The HVAC at the hotel does not share any capabilities (it is not mobile-controllable, its use is restricted to the hotel staff or there is not any HVAC), so this action is removed. The trigger event was to plug-in the smartphone to power source, it was a location event: the socket detected that the smartphone was in the living room. At the hotel, this is done by the main door lock that detects when the guest is inside or outside the room. Furthermore, in the hotel case, guests can share their scenes, which are analysed to gather statistical information. These data, mashed up with user preferences, are used to recommend new scenes to the user. In Bob’s scenario, the alarm clock at 8:00 is an example of a shared scene **statistically-adapted**: since almost all guests set an alarm clock, this scene is proposed and, considering most of them set the alarm at 8:00, this is the proposed time. The smartphone contains several information and preferences about the user, so they can be used to adapt a **generic scene** to him. Moreover, in the smart TV scene, the change of photo at 12:00 is not proposed because the recommender knows that in most of the rooms there is no one at that time in a working day, and thus a scene with that action will probably be discarded or modified. As for the user preferences stored on the smartphone, also information from other external sources can be added in order to refine the proposed scene.

3.1.1 Identification and Location

Analysing the scenario carefully, we can identify two capabilities that every smart object should have. Let us consider, for instance, the following scene:

*When Bob plugs his smartphone into the dock in his office,
then the air conditioner in the office turns on.*

Let us suppose that is the dock which detects the plug in event, not the smartphone. The event triggers the behaviour configured in Bob’s smartphone, which sends a *turn on* action to the air conditioner. This scene uses an event (on plug in the smartphone) and an action (turn the air conditioner on), but to work exactly as described in the sentence both event and action need some additional information. The scene should work only when Bob’s smartphone is plugged into that particular dock, not when it happens into any dock, and, of course, neither when it is another mobile to be plugged into that dock. From this consideration, it is evident that there must be some mechanism that identifies which actors are involved when an event, condition or action takes place. In fact, without dock identification, the previous scene will be:

*When Bob plugs his smartphone into **any** dock,
then the air conditioner in his office turns on.*

While without smartphone identification it will be:

*When someone plugs **any** smartphone into the dock in Bob’s office,
then the air conditioner in Bob’s office turns on.*

Coming back to the assumption that it is the mobile which carries the scenes, then there is no problem in the second case, because the air conditioner will turn on only if the plugged in smartphone does have this scene configured and enabled on itself. On the other hand, the first case is more complicated: when Bob use the dock in his office, the scene will work in the expected manner, but when he plugs in his mobile into another dock, it will not, switching on the air conditioner in his office. In order to avoid these situations, smart devices should provide some kind of identifier. During scene execution, this is especially true for sensors, which should add the identifier when they notify an event to a coordinator. For actuators it is not strictly necessary: since the orchestrator directly controls them, it knows (thanks to the discovery) to which object it must send an action. In both scenes reported above, the air conditioner in Bob’s office always turns on, because the action is sent to that device, which has been previously discovered, identified and configured. During discovery identification is compulsory, as every module must refer to its smart object. Identification is considered among the five main capabilities that a smart object should have, together with sensing, actuation, decision-making and networking ([Hernández and Reiff-Marganiec, 2014](#)).

Looking back to the first scene, we use the position of a device to identify it. We said *the dock in Bob’s office*, and not *the dock XYZ*. In natural language is common to refer to something using some of its properties (colour, size, shape, age,

position, ...) instead of its “proper noun”, especially when there are two or more similar things in the same space. The position is a property which by its nature is unique for every thing. If there are two identical bulbs resting on a table, I can easily refer to one of them using its position, for instance using adjectives such as left/right, above/below. There is an association between identifier and position: given a property, it can be used to determine the other one, under certain conditions. Unlike the identifier, position may change, so a mechanism to maintain the position updated is needed. In an ideal situation, the objects themselves are able to determine their position, but in a real case it is hard to achieve. We can simplify this challenge by admitting a symbolic location and by avoiding imposing limits on precision. In many environments, most of objects are ‘static’ (furniture, appliances, decorations, window and door frames, ...), and thus a symbolic position such as ‘*the desk near the main door*’ or ‘*the air conditioner in Bob’s office*’ are accurate enough to identify an object. In any case, the description of a scene stored in the orchestrator will always use the corresponding identifier to avoid possible ambiguities. The mapping between object’s identifier and its location may be maintained in the object itself, in the orchestrator or in another component. Regardless of where this mapping is stored, we consider the location capability of an object its capability to give its (updated) position to the orchestrator. For static objects, the location may be hard-wired or configured by the user before or during discovery. If a location service is available and it can track an object movement, then it can maintain the objects’ location up-to-date.

Location is also important in generalization, since we can refer to several object placed in the same space. A simple form of generalization is removing this information, obtaining a scene that works in every space that contains the same objects. The initial scene, for example, will be:

*When Bob plugs his smartphone into the dock in (space X),
then the air conditioner in (space X) turns on.*

This scene works in any space where there are a dock and an air conditioner. To work, orchestrator needs to know what is the *space X* (specialization) and to fetch the identifiers for these two objects. Once obtained their identifiers, then the scene can work.

In conclusion, location capability helps the user to identify smart objects and to create portable scenes, besides being able to be used as any other capability to configure and execute a scene.

3.2 System Workflow

Considering the scenario and the operational concepts presented in the previous sections, we can identify three main steps in the workflow: discovery, personalization and interaction. During discovery, the user knows what objects are close to him and what they can do. In personalization, he decides how they should work together. Finally, the objects interact among them, coordinated by the smartphone, according to the user personalization.

Each of these steps can be carried out in some different ways or through sub-steps. All the steps performed to create and execute a scene describe its lifecycle. In Figure 3.1 are arranged all possible steps and sub-steps identifiable from the sections above.

Figure 3.1: System workflow

As previously stated, discovery is the first step of the workflow. During this phase, smartphone detects available smart objects and it acquires their capabilities. Depending on the object, this process can be manual or automatic. In discovery, smartphone acquires a module for each object in which the user is interested within a smart space. The module contains the capabilities (events, conditions and actions)

that an object is able to perform, parameters that increase capabilities' expressiveness, variables that an object can share and a driver that instructs the smartphone on how to interact and control the object. At least one capability and the driver must be present in a module.

In manual discovery, user must perform some form of “interaction” with an object to obtain its module. This is the case of objects that cannot share a module autonomously, because they do not have a storage space suitable for that purpose or they are not able to send the module to the smartphone through a Wi-Fi connection. An external server or additional hardware are used to overcome this shortfall. The simplest solution is to rely on a module server. A server stores the module, while the object publishes an identification code using a QR, a bar code, an NFC tag, an alphanumeric code or other methods. The smartphone uses the code to generate a GET request. Then, the request is used to download the module from the server. Several commercial smart devices currently use this solution. Manufacturers easily share software or services on their public servers, which are accessed by users using a specific identifier associated to the purchased object. Examples of this method are some HomeKit compatible objects that have a numeric code or Philips HUE bulbs that work with a gateway: user must press a button on the gateway to associate the smartphone application with the bulbs and the cloud service.

In automatic discovery it is not necessary any user interaction. Objects announce their presence and the smartphone automatically gets the module. This is the case of many multimedia objects that have a discovery mechanism (DLNA, AirPlay, Miracast, etc.), or other objects that implement UPnP or other protocols with a discovery algorithm. Since few objects are able to store and share a module directly, often an external server is also used in automatic discovery. The process begins when the smartphone connects to a WLAN. When this happen it automatically receives all module that belong to the object in that local network. If a module server is used, smartphone receives the identification codes from the objects (for example the unique device identifier for UPnP compatible objects) and it generates the GET request to download the module.

Modules contain objects' capabilities so, when the smartphone has downloaded them, it automatically gets aware of what the objects can do. Capabilities are mashed up into scenes that describe the space behaviour.

Personalizing a space means assembling capabilities to create scenes. This is the second step in the workflow. Scenes can be created by the user or by another entity. The user can manually create scenes mashing up the received capabilities and following the ECA paradigm. ECA paradigm only imposes restrictions on the cardinality of capabilities of each set and on the order. In a scene, there must be exactly one event, zero or more conditions and at least one action. The execution of the three sets must be strictly sequential: first, the event that triggers the scene, then all the conditions are checked and finally the actions are executed. Also within

the same set of capabilities of a scene, the execution must be sequential.

Execution of scenes is concurrent: when an event is notified, all the scenes that have it are triggered at the same time. It is possible to synchronize scenes by automatically enabling or disabling them during the execution. These are the only prerequisites that the user must keep in mind to create a scene. Some control mechanism can be added to help the user to check the consistency of a scene after its creation (Iglesias et al., 2012). Users can share their scenes with others, providing them a fully functional scene or at least a basic one that will be adapted according to the needs of each user. This adjustment is manually done by the user without any previous automatic manipulation, so this case falls within manual scene creation.

Beyond pure sharing of scenes among users, others more or less complex mechanisms can be used to filter scenes and to generate new ones for a user, in accordance with his preferences or other factors. This is the process of recommendation. The recommenders can be implemented using some artificial intelligence techniques (case-based reasoning, artificial neural networks, agents, etc.), statistical inference or ontologies. Recommendation works in several different ways using previous scenes shared by the user or by others, his preferences and the capabilities available in a smart space. We identify two main ways of work: a user-scene-dependent way that modifies a previous scene shared by the user himself, and a user-scene-independent way that uses statistical information, scenes shared by others and user's preferences. Since a scene is adapted, we call adaptation these two processes. The difference relies on who provides the initial input scene(s) to the recommender, the user or a database of previous scenes. If there is not an initial scene, it is the previous manual scene creation case.

The first one is an adaptation based on similar scenes: the user implicitly provides a scene and the recommender modifies and returns it to the user. To do that, the recommender can use the capabilities available in a smart space or the scenes previously shared by other users. The first case is the most important because it works without other scenes and it represents the process that adapts a user's scene to a new space. The recommender analyses the scene and the available capabilities and returns a new scene that works in the new space (at least in a similar way), or it discards the scene, if it is not adaptable because of the lack of some important capabilities. The adaptation can be improved using scenes shared by others that can address the decisions of the recommender.

The second kind of adaptation is based on user's preferences: the user does not share a scene with the recommender, but he shares some of his preferences. The recommender selects a scene performing a statistical analysis of previous shared scenes and it selects or generates a new one with most used features from the old ones. Then it refines the new scene using the user's preferences (in the scenario, when the hotel propose the alarm clock scene). Preferences can be provided implicitly, obtained from parameters and variables that the user has configured in his previous

scenes, gathered by the smartphone and sent to the recommender; or explicitly, if the user directly provides his preferences through the smartphone application. In order to increase the privacy level, the last refinement may be done directly by the smartphone, in this manner preferences are not shared.

Regardless of the kind of adaptation, the user can always manually modify a scene proposed by the recommender.

After a scene has been created, it is manually activated by the user. From that moment, the scene is active and the generation of its event will start the interaction. Sometimes an object needs a previous configuration, especially if it is the sensor that generates the event. If a scene begins when the temperature in a room is out of a range, and the temperature is registered by a thermometer, then it needs to know that range or it will not be able to generate the event. Without this preliminary configuration, the scene will not work. The smartphone automatically detects this cases and, before actually activating the scene, it sends the configurations to the objects that need it. This step also includes a previous activation or deactivation of sensors that must be prepared to generate events.

After the personalization, scenes are executed: smart objects interact among them, orchestrated by the smartphone. A scene describes a particular behaviour of some objects in a space. Smartphone knows what objects are able to do and how to interact with them, thanks to the module downloaded during the discovery phase. Scenes are ECA configurations. They are activated by an event, it is detected by a sensor and sent to the smartphone. If conditions are present, the smartphone queries the related objects to check them. Finally, one or more actions are launched.

In order to allow the execution of chains of scenes, the object configuration step must also be able to be performed automatically, during the interaction phase. Let us consider the following two scenes: *when the temperature rises above 27 °C, then switch on the air conditioner* and *when the temperature drops below 25 °C, then switch off the air conditioner*. As said previously, if the temperature sensor triggers the event, it must know the value to be able to generate the event. To ensure that both scenes are functioning correctly, there are two possibilities. The first one is available only if the temperature sensor can store more than one value. In this case, during configuration, both 25 °C and 27 °C are sent to the device and both scenes will always be available. The other possibility occurs if the sensor can only store one value at a time. In this second case, it is possible to use both scenes creating a (closed) chain such as: *when the temperature rises above 27 °C, then switch on the air conditioner, disable this scene and enable the other one* and *when the temperature drops below 25 °C, then switch off the air conditioner, disable this scene and enable the other one*. In this manner, both scenes can work, but only one at a time. To work, this behaviour needs to perform a change of configuration of an object during the switch between one scene and the other. The sensor must change *the temperature rises above 27 °C* by *the temperature drops below 25 °C*. This

variation is possible admitting that the capabilities, used during the interaction, are able to modify the object configuration automatically. In general, every time that a scene is activated by another one, it might be necessary to automatically reconfigure the objects for the second scene during the interaction. This reconfiguration not only affects objects' parameters, but it can also turn on and off objects or their sensors and actuators.

3.3 Technology Agnostic Architecture

The sections above present all the possible steps that a user can follow to personalize a smart space. These steps are performed by different components that reside in the smartphone, in a server or in the smart objects. Each step is performed by one or more components. Following the workflow is possible to identify these components and their task in the system, regardless of their implementation or physical location. The components are arranged in the architecture showed in Figure 3.2.

Starting from the bottom, there are the smart objects. Objects communicate with other objects, servers and users. They perform two main kinds of communications: they publish their capabilities to the rest of the system and they take part in one or more scenes, coordinated by the smartphone. Objects can have many different telecommunication technologies and protocols, and thus gateways are needed to ensure a communication between objects in the same subnet and with the rest of the system. Internally, subnets may use any type of networking protocol, for example, an IoT protocol or an ad-hoc one designed for a group of specific objects.

A common bus centralizes and translates all the communications between the objects and the rest of the system. It pushes the capabilities and the object's information published by the smart devices to the resource catalogue and enable a bidirectional communication between objects and scene execution block. The latter type of communication allows the orchestration of the objects during a scene execution.

The location manager also receives the capabilities and the object's information. When it happens, it determines the location of the objects.

Entering into the discovery phase of the workflow there are the resource catalogue, the location manager and the resource filtering. Their task is to find smart objects, to get their capabilities, to determine their position in a smart space (and the smart space to which they belong) and, finally, to organize all the information gathered and to present it to the blocks above.

Every smart object has an associated module. A module contains two main parts: the manifest and the driver. Manifest gathers the definition of the object's capabilities and its parameters and internal data (e.g. the object's identifier, location, owner, accounts, basic configuration, etc.). The driver stores the code to perform

Figure 3.2: Technology agnostic architecture

the capabilities. Modules are compressed files, the resource catalogue receives and processes them, splitting the driver, the capabilities and the rest of data. Two identical objects have different modules, but within the modules, drivers and capabilities are the same, only device's specific data are different.

Resource catalogue has a data structure that contains all the drivers and capabilities from the modules downloaded from all the objects that have been discovered, including those discovered in other previous spaces. If a new object is detected but its module is not available, resource catalogue downloads it from a repository or it sends a petition to the common bus to request the module directly to the object. Only one copy of the driver and capabilities for each identical object is maintained, duplicate parts are deleted, in this manner the scalability is improved: with one copy of capabilities and driver it is possible to control several identical devices. Since object's information are packaged in the module, it is always necessary to download it, even if the driver and the capabilities are already stored in the data structure, to obtain the rest of the data.

Location manager stores the rest of the data with a reference to the capabilities and driver of a given object. Two additional data are always included in a module and they are the most important fields to uniquely identify and locate a device: the

unique identifier and the location (location may be manually added or modified, but it must be present when discovery finishes). The role of the location manager is to maintain all the known objects organized according to their location. It helps the resource catalogue to maintain the organization of the objects into spaces and, if possible, within a space (if a location system is available).

Modules repository is also organized into spaces; the easiest configuration has a repository for each space. When discovery finishes, if the user has accepted and downloaded all the modules, resource catalogue and modules repository have the same modules, for the considered smart space. Location manager also works to maintain the repository updated according to the user position, for instance if a new object is found and not available in the repository, by adding it to the repository, or deleting objects that are no longer available in the space where the user is. Resource filtering eventually aids the user or the recommender to create or propose scenes, hiding or showing them only some objects or capabilities.

Personalization phase begins after discovery and it uses objects and modules that have passed the resource filtering. Manual scene creation is a graphical user interface that permits the manual creation of a scene. Scenes may be shared with other users, storing them in a scenes repository. This repository is also used by the two adaptation blocks. A user's scene, created for another space and stored in the repository, can be adapted for the new space. In a similar way, scenes shared by other users in that space can be analysed and adapted for a new user. Finally, before the execution, a scene compatibility analyser can review a scene looking for errors and inconsistencies. It also has access to the modules repository to check if the modules used are the right ones for that space.

After the personalization, the scenes are executed. Scene execution block orchestrates the objects through the common bus and following the configured scenes.

This architecture only shows the components necessary to work, but it does not describe any internal data structure or how it is possible to control any kind of smart object with the same blocks. In a real implementation, the system needs a data structure able to describe the objects and their capabilities in a manner that the system can handle them easily during each step of the workflow. Moreover, this data structure must be open to allow the development of new objects compatible with the system. This data structure will be proposed in the next two chapters, beginning from a model that describes a generic smart object internally and later modelling any object as a black box able to interact with other objects and controllable through a common interface.

3.4 Summary

In this chapter, we presented the operational concept of a system able to control smart objects. We first analysed, through a scenario, which are the features and the steps that a system must have to control smart spaces. Starting from the scenario, we extrapolated the steps through which the system must evolve, so we got its workflow. It is composed of three main phases: discovery, personalization and interaction. Each step has been analysed.

In discovery, objects publish their capabilities which are learnt by the user's smartphone through different mechanisms. Capabilities are mashed up to create scenes, during the personalization phase. This phase can be done manually by the user or automatically. In the latter case, the system can adapt a user's scene for a new space, adapt a scene shared by other users for that space using user's preferences or propose a scene using a statistical analysis of shared scene for that space. Finally, the scene is executed in the interaction phase, where the user's smartphone orchestrates the smart objects to fulfil the configured scene.

The last section describes a technology agnostic architecture where the components, that perform the steps of the workflow, are identified, regardless of their location, implementation and technologies employed. This architecture shows the need of a data model that enables an easy communication between objects and system during discovery and execution.

Chapter 4

Smart Object Modelling

This chapter is devoted to the modelling of smart objects and smart spaces. We apply some Object-Oriented Programming (OOP) concepts to the modelling of objects and spaces.

One of the original purposes for OOP languages was to model applications that have multiple objects that may be operating simultaneously. Control became distributed throughout the application: at any given moment, multiple objects might want to perform some action. OOP solved this problem by making ‘everything an object’ and having control reside within each object instead of in a single thread. During the execution, objects would communicate with one another by passing messages. A message is simply an invocation of an operation in another object. These were the two main changes introduced in the execution model with OOP: simultaneous execution and exchange of messages.

First OOP languages were created for simulation purpose (that is what Simula I was named after, in 1960s). They were initially used to generate models of factory floors and assembly lines: environments where there are machineries that work simultaneously and that are able to exchange messages using a communication line. Other environments, such as a home, were more difficult to simulate: objects were often simple tools that had to be used by a person (they could not work autonomously) and that could not communicate among them. These assumptions make useless the OOP execution model, and probably even the simulation.

Nowadays, a lot of tools have been enhanced with electronic components, giving them some processing and communication capabilities and making them ‘smart’. This transition from object to smart object pulls together the OOP execution model and the way we use smart objects. In principle, smart objects can work autonomously and simultaneously, and they can communicate among them. Therefore, a program written with an OOP language can describe the interactions between objects that happen in a smart space. As for the execution model, also other OOP

features, such as encapsulation, inheritance and polymorphism, can describe smart objects and the relations between them. To explain the OOP concepts, real life examples are often used. We want to use this duality OOP-real world to model smart objects, and describe and automate their interactions in smart spaces.

This chapter begins with a brief introduction to OOP, its main features and how we will use them to model smart spaces. Then, we describe a generic smart object, focusing on its internal components and on the communication process with other objects for both discovery and interaction phases. Beginning from this description, we extrapolate a class diagram of the generic object. Using inheritance, we extend the generic object model to any physical device. Thus, we manage to describe any smart space in terms of objects that compose it and their capabilities. This model will be used to define which capabilities should be provided by an object during the discovery phase. Reinterpreting some OOP features, such as polymorphism and encapsulation, the model will also be used in personalization and interaction phases. In personalization, defining which capabilities of which objects may be used to create or modify a scene in a given space, or adapting an existing scene, created for a given space, to a new one. In interaction, following the basic OOP execution model: objects call methods exposed by other objects, without knowing their internal structure and implementation.

The model described in this chapter has been created keeping in mind the ‘static’ relation between objects, internal components and capabilities, and the system workflow described in the previous chapter. The latter requires the user’s smartphone to discover objects, to personalize scenes and to orchestrate the interactions among the devices during their execution.

Models analysed in the state of the art ([Goumopoulos and Kameas \(2009b\)](#), [Strohbach et al. \(2004\)](#) and [Parra et al. \(2009\)](#)) was proposed thinking in a distributed system. These kinds of models are inadequate to describe the personalization phase of the workflow. All these models directly embed the domain knowledge in the objects: interactions among objects are hard-coded into the devices themselves, and thus an external entity cannot change it easily or without user intervention. A distributed system prevents a single point of failure, but sacrifices the personalization. Our model exhibits a single point of failure because it needs a smartphone or another controller to orchestrate the interactions, but it provides an easy form of personalization (manual or automatic using the OOP) and ensures the portability and adaptation of a scene to other smart spaces.

4.1 Using OOP to Model and Network Smart Objects

The term “object”, in the modern sense of OOP, made its first appearance in the late 1950s at MIT. A few years later, in 1962, Ole-Johan Dahl and Kristen Nygaard created the first OOP language: Simula I. Five years later they presented Simula 67 which formally introduced the concepts of objects, classes, subclasses, inheritance, virtual methods, discrete event simulation and garbage collection ([Dahl, 1968](#)). In 1972, Alan Kay extended the key ideas used in Simula 67 to create an early version of Smalltalk called Smalltalk-72 ([Kay, 1993](#)). This programming language evolved until it reached a stable release in 1980. Smalltalk-80 added metaclasses, attributes encapsulation, abstraction and polymorphism, to help maintain the “everything is an object” paradigm ([Black, 2013](#)).

The principles introduced by Simula 67 and Smalltalk-80 were becoming firmly established. About twenty OOP languages were born between the 1980s and today, and a further similar number of languages, based on other programming paradigms (especially imperative and procedural programming), have been adapted to work as OOP languages too.

Nowadays, OOP is considered a design philosophy. It is based on the concept of objects, which are structures that contain data (attributes) and procedures (methods). Object’s procedures can access and often modify the data fields. OOP, to be considered as such, must provide at least three main mechanisms that objects may use: encapsulation, inheritance and polymorphism ([Scott, 2009](#)).

Encapsulation is a concealment of implementation: it allows selectively hiding attributes and methods of an object. It consists in the separation of an interface from the related implementation. Thus, objects are black boxes that expose interfaces. An object can interact with another calling the methods of its interface and without knowing its internal implementation.

Inheritance is the ability to reuse the definition of one kind of object to define another kind of object. It allows creating a general-purpose element base class that contains only the fields and methods needed to implement an object. This concept is strictly connected to abstraction and interfaces. A base class describes which attributes and procedures should have an object that implements that class. With inheritance, it is possible to define a graph (UML class diagram) that groups objects in function of their common fields and methods.

Polymorphism is a generic term that means ‘many shapes’. More precisely, polymorphism means the ability to request that the same operations be performed by a wide range of different types of things ([Hunt, 2002](#)). With this mechanism, several objects can do the same thing, so they are interchangeable to do that thing. Interchangeable objects have the same common interface.

[Armstrong \(2006\)](#) analysed the existing literature and she identifies other five concepts, often used when talking about OOP, that should be added to encapsulation, inheritance and polymorphism, in order to define an OOP language. She adds object, class, abstraction, message and method.

Objects are individual, identifiable items, which contain information about themselves and the descriptions of its manipulations. Objects are the basic blocks of OOP (hence its name). A class is a generalised description of similar objects that share a common structure and behaviour. Abstraction is the process of creating classes representing reality in a simplified form by removing certain distinctions between objects. A message is the process by which an object sends information or invokes a method. Methods are procedures, associated with a class, that access, set or manipulate an object's information.

Other concepts, such as association, aggregation, composition, generalization, etc. are used in OOP and will be used in this chapter to create our models.

One of the objective for which OOP was created was to associate real-world objects and processes with digital counterparts. Our idea is to invert this association and coming back to the real objects reinterpreting OOP concepts. Smart objects have an important characteristic that 'normal' objects do not have: they are able to communicate. With this feature, the basic interaction between objects in OOP, an object that calls methods of the other, can be directly mapped in the real world. Thus, smart objects can call methods of others smart objects; they can exchange data and share their capabilities to create complex behaviours.

Table 4.1 briefly describes the three main OOP features with their definition and our reinterpretation to apply them to smart spaces. Our reinterpretation of OOP features will be explained and used in the rest of this chapter end in the next one.

In the following section, we analyse a generic smart object with the purpose of identifying its internal components and creating a model able to describe any smart object.

4.2 Smart Object Model Proposal

Let us consider a physical object. It exists for one or more specific aims (a box to contain something, a key to open and close a door, a thermometer to measure temperature, etc.). The aim is the result that people perceive using or simply observing an object. An object is a mean to achieve a goal: it provides some 'passive' functionalities that a user exploits for his needs. The user usually does the concatenation of two or more capabilities from different objects manually, for example, he opens a door of a dark room and he switches the light on, there is not a direct interaction between door and the light. A smart object is an enhanced version of the corresponding object. It has a communication interface that enables

Feature	OOP definition	Reinterpretation in smart spaces
Encapsulation	It hides complexity. Objects are black boxes that expose only some attributes and methods	Objects are accessible only through their published capabilities
Inheritance	It enables new objects to take on the attributes and methods of existing objects	It allows us to create a taxonomy of capabilities that is used to select the objects in recommended scenes
Polymorphism	Multiple types of objects might be able to work in a given situation	In a given scene, it permits to exchange an object for another one that publishes the same capability involved in that scene
Object	Individual item (real or abstract) that contains information about itself and the descriptions of its manipulations	Physical device able to communicate and to perform some task (smart object)
Class	A description of the organization and actions shared by one or more similar object	A description of a group of objects that share the same capabilities
Abstraction	Mechanism that allows us to represent a complex reality in terms of a simplified model so that irrelevant details can be suppressed in order to enhance understanding	(The same)
Message	An object sends data to another object or asks another object to invoke a method	(The same)
Method	A way to access, set, or manipulate an object's information	Object's capability to interact with the real world, other objects and users

Table 4.1: Definition and reinterpretation of the main OOP features

the direct interaction with other objects and some electronic components that permit an autonomous behaviour or, at least, a remote control.

A smart object usually maintains the same service goals, and thus the capabilities of the corresponding non-smart version. Capabilities are what an object can do. We assume that an external entity may call these capabilities and then control the object behaviour making it do what it wants to do. When an entity calls a capability of an object, it internally processes the call. To do that, the object uses a set of internal processing mechanisms and a set of internal states. Using these two components, the object converts an order into a real behaviour. We name the set of processing mechanisms as **functional scheme**, and the set of internal states as **state vector**. A state vector defines at any time the conditions of the various components of an object (sensors, actuators, etc.). The union of the state vectors of the objects that compose a space is the state vector of the space: it may be used to represent a scene, which describes a behaviour of the space. Selection and configuration of a scene are the basic mechanisms of personalization. The functional scheme is a set of rules that control the transitions between the states of an object. It determines how to change the state vector depending on the data received from the sensors and its internal configuration. In the same way, it decides which actuators are to be controlled and how. It also controls the communications. Functional scheme is the implementation of the capabilities exposed by the object. When an entity calls an object's capability, the object's functional scheme is responsible for executing that capability, changing state vector and controlling its internal components. Objects publish their capabilities during discovery phase (packed into the object's module in the **Manifest** section). Capabilities describe what an object can do. A published capability may be called remotely, so an external entity can control the object that exposes it. In most cases, capabilities depend on object's components that interact with the environment (sensors and actuators). Core's capabilities to control sensors and actuators define the object's capabilities.

In terms of hardware, a smart object may have the following components: **Sensors**, **Actuators**, a **Processing Module**, a **Storing Module/Memory** and a **Communication Module**.

- Sensors are devices that detect events or changes in quantities, on demand or continuously. They propagate events to trigger a scene, i.e. changing a state in a state vector of one or more objects. Sensors are often responsible for checking conditions, when these concern physical parameters of the space.
- Actuators are devices able to change the space or the state vector of another object, propagating an action. Enabling an actuator is the result of changing a state in the state vector. When a state changes, the corresponding actuator will begin to work to bring the physical space to the situation described by

that state. State vector is like a panel of switches for sensors and actuators: sensors change switches, while actuators respond to changes of switches.

- Processing Module is the component in charge of executing the functional scheme. Depending on the complexity of the object, it may be a CPU, GPU, ASIC, microcontroller, simple electronic circuit or a combination of many of them.
- Storing Module/Memory is used to store the state vector: this is its primary task. It may also be used to save historical data (measures, events and evolution of the state vector), configurations, users' preferences, etc. Since in embedded devices often primary and secondary storage are implemented in the same element, we consider both of them a unique component.
- Communication component is the module that allows an object to exchange data with other objects, with the user or with other entities. Two important tasks it should perform are to publish object's module and to locate the object itself within a space during discovery.

Sensors and actuators may be internal parts of an object, may belong to another object or may be objects themselves. Combining sensors and actuators in these three ways it is possible to create complex capabilities that use sets of objects. On the other hand, from the point of view of the model, it is possible to divide objects into smaller components, simplifying their description.

Figure 4.1 is a graphical representation of our smart object model with the communications during the three phases of the workflow. All communications pass through the communication components, except those with external objects when they are used as if they were internal sensors or actuators (for example a television that uses external loudspeakers or a weather station that uses an external temperature sensor). In these cases there are ad-hoc connections, but sometimes they can also use the communication components. In Table 4.2 are gathered all the object's components and their role during each workflow phase.

A smart object publishes its capabilities during the discovery phase through its module. These represent the interface between the object and the rest of the space. The use of the capabilities is the only way that an external entity has to access, albeit indirectly through the functional scheme, the state vector of an object and then to use it and to take advantage of its services.

Capabilities are events that an object is able to trigger, conditions it can check and actions it can perform. The user uses these capabilities to build scenes and orchestrate their execution employing the ECA paradigm. Personalization can be potentially produced in several ways: the user can configure a scene, the user can interact with the single objects, or the space can suggest scenes. Some objects, due

Figure 4.1: Components of a generic smart object

to their hardware limitations, cannot share their module themselves. If this happens, an external server can take the place of the object publishing its module.

In order to overcome the limitation of the different protocols and communication technology that objects have, an external entity (user's smartphone) coordinates the interactions between two objects, during ECA interactions. Despite this, direct communications between two objects using a special-purpose link is also possible. An HVAC system, for example, can use an external temperature sensor to control and maintain the temperature in a room. HVAC and sensor can work together, conversing through a dedicated wire, but they can also behave such as independent devices. These communications with external objects are also useful to model complex smart objects, separating them into smaller ones.

Objects can communicate with other objects, with the user's smartphone and with servers. As well as publishing its capabilities during discovery phase, an object can be configured during personalization and controlled during interaction. In personalization, it can receive some parameters or configurations that it will use in the next phase. A temperature sensor, for example, may be configured to notify when the temperature in the room rises above 25 °C. In general, sensors need a configuration to determine how they work, while actuators can receive a configuration during a scene execution. An HVAC can receive the order to maintain the temperature at 25 °C when it is activated, but the temperature sensor cannot notify that the temperature has risen above 25 °C without knowing that limit before the scene activation. During personalization, other entities may also request data from

Component	Workflow	Role
Sensors	Discovery	–
	Personalization	Receive and set a configuration for themselves
	Interaction	Detect or measure physical properties from the environment; trigger events (modify the state vector); check conditions
Actuators	Discovery	–
	Personalization	Receive and set a configuration for themselves
	Interaction	Modify a physical property of the environment; execute actions (read the state vector); check conditions
Processor	Discovery Personalization Interaction	} Executes the functional scheme
Memory	Discovery	Stores the object’s module and its identifier
	Personalization	Stores configuration parameters
	Interaction	Stores the state vector
Communication	Discovery	Publishes object’s module (capabilities and driver) and its identifier
	Personalization	Receives configuration parameters
	Interaction	Forwards events, conditions and actions between processing modules and the orchestrator

Table 4.2: Components of a smart object

an object, for example to know its current state, configuration or historical data.

4.2.1 Modelling a Smart Object

The model proposed can be used to describe every smart object. In the next three subsections, we apply it to the description of three smart objects: a video wall, a door and a drone. These objects have been chosen to show how we can model devices very different from each other, that have sensors and actuators and that are a composition of other smart objects. These three devices will also be used in the

next chapters to explain concepts through examples and to validate the system.

4.2.1.1 Video Wall

Let us consider a video wall. A video wall is a multi-monitor setup that consists of multiple monitors tiled together contiguously in order to form one large screen. A video controller takes care of splitting, organizing and dispatching the incoming data through the various monitors. We consider a video wall able to receive a content through a network interface, to change the multi-monitor configuration and to send a different content to each tile. Monitors are simple displays which show a video provided by an external source and they cannot be used without the controller.

Figure 4.2: Model of a video wall

In Figure 4.2 there is the model. It is composed of several screens and an object core that is the video controller.

Each physical screen is an actuator, which is only able to show the content that

it receives through its input source. The core separates a video content into nine different portions and sends them to the screens. Core has its internal memory and processing unit. The state of the object is determined by the core's state vector, which has five fields. First field is `id` that contains the object's identifier, that is used by external entities to identify the device. Second is the object's position. Since a video wall is a big object with a reduced mobility, its location is hard-wired during its installation to have the value `room1`, which represents the symbolic room name. Next field is `content`, it contains the content to display. In this example it is a video (`surf.mp4`), but it may be a photo, a shared desktop or a combination of various contents to show simultaneously. In the latter case, it is the `config` field that determines the current tiles layout. For instance, if the video wall must behave as a single screen (of 3x3 screens), if each tile must work as a separate monitor (nine tiles of 1x1), or any other possible configuration (one tile of 2x2, one below of 1x3 and one on the right of 2x1, etc.). Core's functional scheme receives the contents to show from the network, splits them (if needed) in accordance with the `config` field, and finally it sends the right part of the contents to each monitor. The last field is `isWorking`. A Boolean value which defines whether the entire video wall is working or not. It works like a general switch for the object, turning it on and off via software. Therefore, if it is set to true, any new configuration or content sent to the video wall will take effect immediately.

In this example, we suppose a very simple video wall with a limited number of capabilities. Capabilities come from object's sensors, actuators and from its ability to process and store data. They are listed in the module's manifest. We suppose the video wall is able to give its identifier, to give its position, to change the tile layout, to play one or more contents (even simultaneously) received from the network and to stop playing it/them. The module also contains the driver that instructs the orchestrator on how to use the capabilities. Module is published by the object and received by the orchestrator during the discovery phase. All communications between the video wall and other entities pass through its network interface.

In the instance shown in Figure 4.2, the video wall is playing a video (`surf.mp4`), sent by a smartphone, on a single 3x3 tile. The video is automatically split by the controller, which sends the right portion of the frames to each monitor.

4.2.1.2 Door

The second smart object we consider is a smart door. A smart door is a normal door with some enhanced capabilities. In this example we consider a door able to detect its state (open or close) through a magnetic sensor installed on it. Moreover, it has a smart lock with a NFC reader for access control and it can be locked or unlocked remotely. The door has a network interface that allows a bidirectional communication with it. We can consider the door a composition of two main complex objects (the

door itself and the smart lock) and several sensors and actuators (magnetic sensor, NFC reader and the latch bolt's motor) connected to the two main cores. Smart lock and smart door can work as standalone objects, in fact, they both are smart objects and they have all components to work without the other. However, in this example, we want to consider the door as a composition of two smart objects that work together. Therefore, we imagine that the smart lock has its communication interface disabled, and it is connected to the smart door's electronics which handles the entire object and the communications with the external world.

Figure 4.3: Model of a smart door

In Figure 4.3 there is the representation of a smart door, described using the proposed model. It is composed of the two aforementioned main blocks: the door's core and the smart lock, that control the rest of the components. The door's core is directly connected with the network interface, and thus it takes care of the communications with external objects and of dispatching data to the internal components. It has a magnetic sensor and it sees the smart lock as an external sensor and actuator.

Its state vector contains the object identifier, the position and two other fields that describes the magnetic sensor. The object identifier `id` will be used by other external entities to uniquely identify this smart door. The `position` field describes the door location, in this example it is the `maindoor`. The other fields directly influence the

magnetic sensor: **isSensing** determines whether the value read by the sensor must be considered and propagated to external entities or not, while **isOpen** represents the actual state of the door (open or close).

The smart lock can work autonomously, reading a NFC tag and locking or unlocking the latch bolt in accordance with a list of authorized tags. It does not have any (public) communication interface, it only has an ad-hoc connection with the core, so it needs to use the core to communicate with external devices. As mentioned before, its network interface is disabled (marked in gray).

Its state vector has seven fields. Since it cannot communicate with the outside world, **id** and **position** are disabled. If a device requests these values, the door's core will reply with its fields. In this manner, both the lock and the door are seen as a single object. **isSensing** and **isWorking** enable the NFC reader and the latch bolt's motor respectively. **isLocked** stores the smart lock state (locked or unlocked). **token** temporally saves the value read by the NFC reader. When a new value is available, it is compared with each value stored in the **accessList** that contains the authorized tags, if the value is in the list, the latch bolt is unlocked. After this check, **token** becomes empty.

Regardless of its internal implementation, the smart door is seen, from the outside, as a unique object. Its components define a list of capabilities, that are published all together in a single module, with a single manifest and a single driver. In this example, the door publishes seven capabilities, three derived from the door's core and four defined by the smart lock (red dotted lines point out which capabilities are defined by each object's internal component).

The first two are the capabilities to give its identifier and its position. The following is related to the magnetic sensor, which detects whether the door is open or not. The next four capabilities derives from the smart lock: the latch bolt's motor can lock and unlock the door, while the NFC reader is able to read NFC tags and to configure the access list. A capability represents a 'service' that an object is able to provide to another entity. Therefore, the value of a NFC tag and the door state may be sent to external objects.

4.2.1.3 Drone

The last example we consider is an unmanned aerial vehicle. Let us imagine a drone, composed of four rotors, equipped with a payload that facilitates retrieving video and able to fly through a series of waypoints using a GPS module. It also has an inertial measurement units (IMU), with a gyroscope and accelerometers, used for navigation and stabilization purposes.

The model is shown in Figure 4.4. This drone is composed of two main subsystems that control the physical components: one for the camera and one for the drone itself. Each subsystem has its own sensors and actuators. Drone's core is directly

Figure 4.4: Model of a drone with camera

connected with the rotors, with the GPS module and with the IMU. Moreover it sees the camera as a sensor. IMU is marked in grey because it is used by the core, but data generated by these sensors are not available to the rest of components, neither to external entities. Its data are only available to the core for navigation and stabilization. This is an example of hidden sensors that exist but cannot be used for purposes other than those for which they were originally intended.

The camera has its imaging sensor used to capture photos and videos. Both sub-systems have their own processing and memory units, but only the drone has a network interface. The camera is hard-wired to the drone's core, which works as a proxy when the camera wants to send captured data to a remote device.

Core's state vector has eleven fields, of which four are dedicated to the GPS and four to the rotors. `id` stores the object's identifier. Differently from the other two examples, this time the core does not have a `position` field. The reason is that the drone can move, so its position may change quickly. Moreover it has a GPS, which is able to provide the drone's current position, making another `position` field useless. The second field is `isWorking` that defines the current state of the drone (on/off). The third is a vector named `wayPoint` which stores a list of way points. It is used if the drone wants to fly using this mode of operation. The following four

fields represent data related to the GPS module. `isSensingGPS` defines whether the module is working or not, while `coordX`, `coordY` and `coordZ` represent the current geodetic position detected by the GPS. In this example we imagine four rotors with an electric engine for each of them. Each engine is controlled by the core which has a numeric value of rpm, stored in the `speedR` fields of the state vector. Core's functional scheme converts these numbers in the corresponding voltage value for the engines.

The camera's state vector has four fields. As in the door example, `id` and `position` are disabled: they inherit the core's values because the camera is a part of the drone. If an external object requests these values, the core will receive the request and reply with its values. Any request for `position` will be replied with the position detected by the GPS. `isSensing` enables and disables the camera, while `frame` is used to save the recorded video and eventually to stream it to external objects.

The drone publishes a module with a manifest that contains seven capabilities. As for any other object, it is able to provide its identifier to external entities. GPS supplies a real-time value of drone's location. It is possible to configure a list of way points through which the drone will pass. It can be remotely moved sending commands in real-time. Last three capabilities are related to the camera. It can start and stop recording video and eventually stream it to another object. Finally, it can take pictures and send them to external devices.

In the Figure 4.4, the UAV is flying toward the waypoint with coordinates 40.4, -3.81, 750, each of its rotors are working (rotor1 at 4000 rpm), camera is streaming its video to a smartphone and the GPS is on, reporting the coordinates 40.404, -3.836, 753.

4.3 Formalization of the Generic Object Model

The model proposed above derives from the observation of smart objects, their internal structure and the adaptation of the Von Neumann architecture to mobile and ubiquitous devices. Applying OOP concepts to that model, it is possible to obtain a formal class diagram. It can be used, reinterpreting some OOP features, to personalize, adapt and recommend scenes. In this section, we formalize the generic model of an object, converting it into a class diagram using UML. In Chapter 5, we combine the class diagram of various objects, the relations among them and the reinterpretation of OOP features to achieve personalization, adaptation and recommendation.

The model has been drawn thinking in the interaction between objects and orchestrator. For this reason, some parts have the purpose of exchanging data with the smartphone, not (only) of working for the object itself. In this section, these parts are described as part of the model, but their function will be more clear in the next

chapter, where they will be used for the interaction.

By observing the Figure 4.1 and the three examples proposed, we can describe the smart object's core using a class, where attributes represent the state vector and methods represent the functional scheme. A smart object may contain sensors and actuators, which are the components that interact with the physical world. They also have their attributes and methods.

Figure 4.5 shows the model, described using a UML class diagram. We model the

Figure 4.5: Generic smart object described with a class diagram

core with an abstract class named **SmartObject**. It will be the base class for every smart object's core. This class has four attributes. The first two are **id** and **position** which stores the object's identifier and its location respectively. The visibility of these attributes is protected: a smart device that inherits from **SmartObject** will assign its own values to them. The last two attributes are two collections of sensors and actuators that compose the object. These collections can be modelled as arrays of abstract classes **Sensor** and **Actuator**. Their visibility is public to allow to other entities to call their methods.

SmartObject's methods are devoted to data elaboration, storing, control, configuration and communication with other entities. All of them are abstract, so a subclass of **SmartObject** must provide its implementation. The first six are protected and they are intended to be internally used to access to the processing unit, the memory and the network interface. A subclass can change the visibility of them to provide some additional service to other entities, apart from adding its own new methods. The capability of treating and elaborating data is represented by the method **process()**. **read()** and **write()** characterise the capabilities of retrieving and store data into the memory module. **send()** and **receive()** enable the communication

with external entities through the network interface. `configure()` also uses the network interface and it represents the capability of setting up all object's components during the personalization phase, prior to the execution of a scene (see Section 3.2). Every smart object must provide the last three methods: `publish()`, `getId()` and `getPosition()`. They are used, during the discovery phase, to identify, locate and know what an object is able to do. Moreover, `getId()` and `getPosition()` may be also used during the rest of the workflow. These two methods return the value of `id` and `position` respectively, while `publish()` returns the object's module, which contains a driver and the list of the object's capabilities stored in the manifest. In Subsection 3.1.1, we introduced the duality identification-location, so, sometimes, one of this two fields may be used to obtain the other one, during one or more phases of the workflow.

As we explained previously, discovery phase may be carried out in two ways: automatic or manual. In the automatic discovery, objects and methods behave in a standard OOP manner: someone invokes an object's method and the object itself handles the call, executes the operation and returns something (or not). On the other hand, in manual discovery we consider smart devices that cannot perform this task autonomously. In both cases there is a reinterpretation of the OOP execution model.

In automatic discovery, we suppose that a method can be invoked on a group of objects in a "multicast manner": making only one call to a set of devices. Discovery process is the way the orchestrator knows the objects around it and what they are able to do. That means, orchestrator calls `getId()`, `getPosition()` and `publish()` on all the reachable objects and it gets their identifiers, positions and capabilities. Ideally, an object may even broadcast these information periodically, without the need of asking for them (e.g. UPnP).

In manual discovery, we admit that a device, often with limited hardware, can modify or delegate the execution of these three methods, as long as the result of the entire process is the same (orchestrator knows at least object's identifier and capabilities).

Standard OOP execution may change in different ways.

- When a single method invocation is sent simultaneously to a group of objects (automatic discovery).
- When no method is invoked because a smart device broadcasts its information (automatic discovery).
- When an object's identifier, location and/or module is provided in an out-of-band manner (manual discovery). For instance, if an object's id is stored on a QR code, then calling `getId()` means: the orchestrator scans the code and obtains the identifier.

The execution may also be delegated to another entity. In some configuration, a server may store information about the objects. An invocation of a method will be redirected or directly sent to the server, which will provide the information to the orchestrator. This solution allows to simplify the objects by ensuring they do not need to store and provide a lot of data, especially the module.

In brief, `getId()`, `getPosition()` and `publish()` intervene during the discovery phase and their purpose is to provide the object's identifier, position and module to the orchestrator, regardless of how the method is 'called' and of who performs the task and returns the corresponding piece of information.

It is evident that the ultimate goal of a smart object is to interact with the real world, for this reason it has been created. Sensors and actuators are the physical components that enables the interaction between a smart object and the environments; therefore they are very important components of a smart device. In the model, they are two abstract classes, and the relation between `SmartObject` class and each of them is aggregation (they can exist independently and they work for the smart object). Physical sensors and actuators inherit from these classes and are collected in the aforementioned `sensor` and `actuator` arrays. `Sensor` and `Actuator` provide five basic capabilities and a control attribute. All methods are abstract, and thus every subclass must provide its own implementation for them and it may change their visibility. Depending on the smart object, its sensors and actuators determine the object's capabilities. Most of them belong to a specific kind of transducer, but some of them are common to all sensors and actuators. These common capabilities are exposed by `Sensor` and `Actuator`, while the other specific methods are exposed by subclasses of `Sensor` and `Actuator`.

Sensors are components that receive signals from the physical world, convert them into electrical output and finally pass the result of the conversion to other components. They can acquire data in four different manners ([Alsbou et al., 2011](#)):

1. **query-based** or on-demand, when an external entity requests the current value to the sensor;
2. **time-driven** when the sensor periodically measures/acquires data, the period between two successive acquisitions may be zero, which means it generates continuous data;
3. **event-driven** when the sensor detects a change of state or when a preconfigured threshold is exceeded;
4. **hybrid** when two or more of the previous cases are combined.

The first three acquisition modes are represented in the model with the methods `getValue()`, `getContinuousData()` and `setEventCallback()` respectively. The

presence of all of them in the same class allows to combine them and acquire data in the hybrid way. Since we use the ECA paradigm in the interaction phase of the workflow, which is an event-driven paradigm, `setEventCallback()` is the most important of the three methods. For this reason its visibility is public and all sensors must have this capability. It intercepts an event and it is able to propagate it to the orchestrator, that will activate the related scene. It is used to set up a callback function on the orchestrator during the configuration phase. It takes a parameter that is a conditional clause (e.g. `temperature > 25 °C`), when the clause becomes true, the callback function is invoked. The clause may be complex, but the sensor must be able to evaluate it (see Section 4.3.1). To simplify the model, there is only one single callback function on the orchestrator: all the sensors call it and they send to it a unique identifier that permits to the orchestrator to know which clause has been satisfied. With this solution, the callback function can be hardwired in the `setEventCallback()` method, and only the conditional clause must be passed, simplifying the configuration process. On the other hand, `getValue()` and `getContinuousData()` work in the standard way during interaction phase: the orchestrator calls them directly and it gets the data. Their visibility is protected because not every sensor must have them. Sensors that have them may change the visibility to allow the orchestrator to use that capabilities. These methods are mainly used to check conditions and to acquire data for other entities.

Sensor class has a protected Boolean attribute `isSensing` that defines whether the sensor is active or not. A sensor is active when the signals it is perceiving are propagated to other entities, not when it is acquiring data. This is because some kind of sensors cannot be turned off, so this field works as a virtual switch, turning off the sensor if possible or not propagating its events and data if it cannot be switched off. Methods `startSensing()` and `stopSensing()` modify the value of the attribute `isSensing` and therefore enable and disable the sensor to propagate the event or the acquired data to the orchestrator. These two capabilities are marked as public because all sensor must provide a manner to enable or disable it.

Actuator class is the dual of the **Sensor** class. It models the components that directly interact with the physical world. As for the **Sensor** class, it has a Boolean attribute `isWorking` that describes whether an actuator is working or not. An actuator start working when the orchestrator calls its `start()` method. Working means performing an action during the interaction phase of the workflow. Every actuator is able to perform an action, so this capability is public. Its dual method is `stop()` which is used to stop the actuator. Since not every actuator can be controlled or forced to stop performing an action, its visibility is protected and it will be exposed only by **Actuator** subclasses that can do it. Unlike sensors, actuators are not reactive components: it is the smart object or the orchestrator who decides when to interact with the real world, not the real world that influences the component. Therefore, `isWorking` reflects the real state of the actuator: if its value is true, then it is actually

performing an action. Moreover, actuator can modify the values of the attribute itself, when it begins or finishes a task. Methods `setActionStartedCallback()` and `setActionFinishedCallback()` notify these changes of the attribute. They are protected methods, so not all actuators provide them, and both of them trigger a preconfigured callback on the orchestrator. `setActionStartedCallback()` notifies when the actuator actually begins to perform an action. It is useful for actuators that need a time between the `start()` command and when they actually begin to work. `setActionFinishedCallback()` works in a similar way when the orchestrator sends a `stop()` order or when the actuator finishes its task. It notifies when an actuator stops performing an action and its `isWorking` field has been set to false. `Actuator` has a fifth method named `control()` that permits to control an actuator in real-time when it is working.

In the model are reported method signatures, but they do not have either parameters or return types. They are intended to be as generic as possible. Return type is not part of the signature, as in the case of many OO computer languages, and it may be typified following the covariance principle (Howard et al., 2003). Input parameters also follow the covariance principle.¹ In sensors and actuators, only `getValue()` and `getContinuousData()` have a return type, while only `control()` has an input parameter. In the following two subsections, we represent, in the figures, some sensors and actuators from objects that we used to test the model in our laboratories. The model can be easily extended: new classes may be added, including their definition and hierarchy in the objects' modules.

The type of the fields reported in the next subsections is different for each class, depending on the kind of data they are able to handle. The type of the fields has the same name of the variables with the first letter capitalized (e.g. `coordinate: Coordinate`) and all kinds directly derives from the generic class `Object` (following the name of the common base class used in many OO languages). To simplify the data exchange between modules, in the real implementation, all fields and values are converted into strings. Complex data such as audio, video and image are stored in files and a string that contains the path of the file is used for data exchange.

4.3.1 Modelling Sensors

Sensors are components that detect events or changes in quantities in its environment and provide a corresponding output. The abstract class `Sensor` only collects some methods that all sensors should have, but it does not describe what physical quantity or data a sensor is acquiring. In fact, this value is what determines the type of sensor

¹Due to the limitation of the programming language used for the development, this feature has been implemented with overloading in the real system.

and, the abstract methods `getValue()`, `getContinuousData()` and `setEventCallback()`, elaborate and return data related to that physical quantity. For instance, a `getValue()` called on a temperature sensor returns a temperature value, `getContinuousData()` provides periodic temperature readings and `setEventCallback()` notifies when a temperature threshold is exceeded. Hence every type of sensor can be modelled with a subclass of **Sensor** that adds at least one specific attribute which determines the class of sensors. These subclasses also assign a default behaviour for the methods inherited from **Sensor** using their specific attributes. The extension of **Sensor** class is shown in Figure 4.6. We use the packages to group and organize classes of similar sensors. The classification shown in the Figure has been done to include most of sensors available in common smart object, despite this, it is impossible to cover every possible sensor (Wikipedia (2016) reports more than 320 kinds of sensor) therefore the model is open and new classes and packages may be added to incorporate new types of sensors.

Location package has two classes of sensors classified according to their location

Figure 4.6: Extension of the **Sensor** class

output. Hightower and Borriello (2001) identify two main types of location for ubiquitous computing: `PreciseLocation` and `ProximityLocation`. GPS and sensors

for indoor localization belong to this package.

Identification and recognition include three types of sensor to cover knowledge (**Pattern**), inherence (**Biometric**) and possession (**Token**). They represent the three main factors described in the multi-factor authentication guidelines for IT security (Gorman, 2003). These sensors can identify and recognize users or other smart objects. A fingerprint scanner and an NFC reader both fit in this package and they work on users and NFC tags respectively.

Environmental package gathers sensors that control environmental parameters such as temperature, humidity, sound, etc.

Space configuration controls the state of doors, windows, blinds and other devices that can change the shape of a smart space, such as movable partitions or curtains. Motion include those sensors that detect movement, acceleration and orientation, such as inertial sensors or compasses.

Medical package has sensors devoted to measure biometric parameters, such as pulse oximeters, ECGs, glucometers, etc.

Finally, virtual package gathers software that use information available from other components to estimate a quantity or detect an event (Liu et al., 2009), they are not physical sensors, but they produce an output similar to that of a sensor and they can work in the virtual world. Examples of virtual sensors are detectors of incoming calls, SMS, mails and other virtual messages.

All the subclasses of **Sensor** are also abstract classes. Therefore they do not provide any implementation of their methods. Physical sensors are modelled with concrete classes that inherit from the subclasses of **Sensor**.

In Figure 4.7 there is an example of a very common temperature sensor (LM35 Texas Instruments (2016)) modelled with a concrete class. It is important to note

Figure 4.7: Model of a LM35 temperature sensor

that the LM35, once powered up, only converts the temperature acquired into an analog electrical signal. The rest of capabilities shown in the class are provided by the smart object on which it is mounted or even by the driver installed in the orchestrator. The sensor cannot perform a **stopSensing()**, but the smart object or the driver installed on the orchestrator can stop propagating the signal to the

orchestrator’s core. Despite this, the capabilities are represented in `LM35` class because they are associated with the physical sensor: each method uses the data that it provides. If it is removed from the smart object or it breaks, those capabilities will no longer function. On the other hand, orchestrator’s core can use the capabilities, regardless of where it is physically implemented their logic. In this example, all methods are implemented and set as public, so the orchestrator can use all of them, while `temperature` field provided by `Thermal` class may be used in `LM35`, but it is not exposed for a public use.

OOP covariance is applied to the methods `getValue()` and `getContinuousData()`. In `Sensor` class they return a generic value (`Object`). In `Thermal`, the two methods return a `temperature` value of class `Temperature`, with `Temperature` $\triangleright$ `Object` (the symbol $\triangleright$ denotes the inheritance arrow). An example of how `getValue()` method changes along the inheritance chain is shown in the Table 4.3. Note the change of the return type that follows the covariance principle.

LM35	<i>Thermal</i>	<i>Sensor</i>
<pre>public Temperature getValue() { Voltage v = readRawVoltageValue(); Temperature t = convertVtoC(v); return t; }</pre>	<pre>public abstract Temperature getValue();</pre>	<pre>public abstract Object getValue();</pre>

Table 4.3: Definition and example of implementation of `getValue()` along the `LM35`’s inheritance chain.

4.3.2 Modelling Actuators

Actuators are devices by which a control system acts upon an environment. Unlike sensors, they are reactive objects: they respond to stimulus from the system, not from the environment. As for the sensors, we try to classify actuators according to the physical quantity they are able to change. For actuators the classification is harder because the physical quantity may not always be clear-cut. A ventilator, for example, does not actually change any physical quantity but the “state of the air”.

The `Actuator` class only exposes one public method (`start()`) that enables the actuator. Its other methods may be implemented and exposed by subclasses. Figure 4.8 shows the UML diagram related to the actuators. As for the sensors, we use packages to group similar kind of actuators and we only include actuator that can easily be found in everyday smart object but leaving the model open to new extensions.

We identify four main groups of actuators: presentation, environmental, machine and virtual.

Presentation collects all those actuators that can show something to the user through

Figure 4.8: Extension of the **Actuator** class

video, audio, images, text and multimedia in general. Screens, displays and loud-speakers are physical actuators that belong to this category.

Environmental actuators are able to modify environmental parameters such as light, temperature or humidity.

Machine package includes motors and engines: actuators directly related to some kind of movement.

Finally, virtual actuators are the dual of virtual sensor. They are software or electronic components able to operate in the virtual world, for example sending e-mails or making phone calls.

Since the argument type is part of the method signature, `control()` is overridden in every subclass of **Actuator** using the type(s) specific for each of them. In **Actuator**, the argument type of `control()` would be as much generic as possible. All subclasses of **Actuator** are abstract classes that only group actuators that act on the smart space in the same way. They may define the capabilities and the fields that an actuator must or should have. Physical actuators are modelled with concrete classes that inherit from the subclasses of **Actuator**.

Let us consider the lighting actuator of a LIFX Color 1000 smart bulb (LIFX, 2016). This bulb is a smart object with a network interface, a microcontroller and an actuator which is a 16 million colours LED light bulb. The model of this actuator is shown in Figure 4.9. It is a lighting actuator; therefore it derives from `Lighting` class. Since it is a multi-colour adjustable-brightness LED bulb, it uses the two protected attributes that describe its current colour and brightness. Regarding the capabilities,

Figure 4.9: Model of a LIFX Color 1000's LED bulb

`LIFXColor1000` class implements only `start()`, `stop()` and `control()`, while it hides `setActionStartedCallback()` and `setActionFinishedCallback()`, not implementing them. It happens because we suppose the bulb cannot give a feedback when it is switched on and off: when its `isWorking` field changes, it does not report this change to the entity that has called `start()` or `stop()`. For this actuator, `start()` and `stop()` assume the meaning of turning on and turning off respectively. The value of `colour` and `brightness` are modified using a setter method that in this case is `control()`. The method `control()` of `Actuator` describe the capability to command an actuator in real-time. Modifying the colour and the brightness of the bulb are exactly this capability. Using a function overloading, we obtain two different implementations of `control()`, that have signature `control(colour)` and `control(brightness)`, and that modify bulb's colour and bulb's brightness respectively.

Covariance is also applied to model the method argument type, not only to the return type. Method `control()` in the class `Actuator` receives a generic argument of type `Object`. In `Lighting`, this type is replaced by `Colour` and `Brightness` which both derive from `Object`. Argument type covariance is not type safe, controversial (Castagna, 1995) and very few programming languages allow it (only Eiffel among mainstream languages). We admit it in the model to maintain the hierarchy, having objects of subclasses more restricted than objects of their superclasses, but, due to the limitations of the programming languages, in the real implementation of the modules, we used the overloading of the methods instead of the overriding.

4.3.3 Events, Conditions and Actions in the Model

In this chapter we used methods and capabilities as synonymous: what an object is able to do is represented with an operation in the class diagram. There are two types of capabilities: those that configure and modify an internal state of a smart object and those that allow the interaction with the physical world and with other smart devices. Looking at the interaction phase of the workflow, we use the ECA paradigm to create scenes and to execute them, so events, conditions and actions belong to the second category, while other capabilities belong to the first one. Hence the first group of capabilities is a sort of complement that prepare the objects to execute a scene, while the second group composes and performs the scene. Events, conditions and actions require an interaction to be completed, so they can be performed only by sensors and actuators: sensors generate events, actuators execute actions and both of them can check conditions. Conditions may actually be checked also by smart objects, without using sensors and actuators, when they refer to the internal state of an object or component. The particularity of conditions is that they always return a boolean value true or false. The output of an event is a notification: a pulse that activates a preconfigured function. An action modifies the environment or the state of an object in relation with the environment.

The class diagrams of sensors and actuators gather some capabilities that these components may (protected) or must (public) have.

Sensor must have `startSensing()`, `stopSensing()` and `setEventCallback()`. To which group of capabilities do they belong? `startSensing()` and `stopSensing()` enable and disable a sensor respectively, so they prepare it to generate an event or not, and thus they belong to the configuration group. `setEventCallback()` represents the trigger propagation to the callback function, so it symbolizes the event itself. `getValue()` and `getContinuousData()` return a value or a continuous signal from a sensor, so their results may be employed to check a condition or they may be stored and shared (see variables in [3.1](#)) or used for other purposes (logs, off-line analysis, etc.).

Actuators must have only a `start()` capability: it is an action. Also `stop()` and `control()` are actions that may be present or not and that stop an actuator or control it during the execution of a task. `setActionStartedCallback()` and `setActionFinishedCallback()` notify when an actuator has actually begun to work and when it finishes, so they are events.

4.3.3.1 Events from Actuators and Actions on Sensors

A sensor is a component that generates events to trigger a ECA scene, an actuator is a component that performs an action. In the class diagram there are four methods that scramble the role of the components.

Calling `startSensing()` or `stopSensing()` on a sensor during the execution of a scene means modify its state, enable or disable it, that is performing an action on a sensor. Therefore, these capabilities may be used to configure a sensor before the execution of a scene, but also to activate or deactivate it during the execution, consequently they belong to both groups of capabilities. Also `getValue()` and `getContinuousData()` may be used as actions on sensors (as well as conditions). `setActionStartedCallback()` and `setActionFinishedCallback()` are the dual case for actuators: they notify when an actuator actually starts to perform an action or when it finishes. They are events generated by actuators. There are not equivalents of these two methods for sensors because when they are enabled their feedback is the generation of an event, and when they receive a stop order, any new event that could be generated between order and actual stop is ignored by the rest of the system (smart object and orchestrator).

In conclusion, all these four capabilities may be used in to configure an ECA scene, and more in general, all capabilities described in **Sensor** and **Actuator** class are events, conditions and actions suitable for describing a scene.

4.3.3.2 Complex Capabilities

The model proposed tries to go deep in the hardware of the objects until sensors and actuators and their interaction with the physical world. This vision is always applicable to the any capabilities, since even the most complex ones can be divided into a processing part and an interaction with the environment. A Kinect motion controller, for example, is composed of an RGB camera, a depth sensor and a multi-array microphone that works together to provide a full-body 3D motion capture, facial and voice recognition ([Kinect, 2012](#)). This system can be modelled separating its sensors and using the `process()` method from **SmartObject** class to create new capabilities that describe the Kinect ones. In fact, it is possible to assembly the Kinect sensors in a new hardware and use a processing unit to create a similar system with the same capabilities.

Even considering this possibility, sometimes can be useful to define a capability directly in a smart object, instead of in sensors and actuators. A good example is a television able to play video on the screen and audio using the loudspeakers. If the content to play is a video with sound, it is possible to describe the play capability as a composition of play video and play audio executed simultaneously. However, this capability can be directly included in the television class. The encapsulation principle perfectly hides playing video on the screen, playing audio on the loudspeaker, handling of the content and the synchronization of audio and video behind a single play capability that automatically uses all these “sub-capabilities”.

4.3.4 Modelling a Smart Object

Figure 4.6 and Figure 4.8 describe the extension of sensors and actuators, while the thermal sensor and the bulb examples show how to model physical components. These components are part of a smart object which is modelled extending the `SmartObject` class shown in Figure 4.5. As for the sensors and actuators, a physical smart object implements the methods that it inherits and it can decide which of them to expose or to hide. Moreover, it can add new methods and it can use its sensors and actuators.

Next three subsections describe how to convert the smart objects modelled before (video wall, door and drone) into a class diagram. From the class diagram of each object its manifest is obtained. The manifest contains the list of public object’s capabilities. These capabilities are the only way that the orchestrator has to control the object (encapsulation). In the manifests in the following examples, capabilities are preceded by one or more letters that marks during which phase of the workflow the capability may be used. In Table 4.4 are reported the letters and their meaning. Lowercase letters indicate a workflow phase with the exclusion of the interaction, whose capabilities are characterized by uppercase letters.

Letter	Meaning
d	Discovery
p	Personalization
E	Event
C	Condition
A	Action
V	Variable

Table 4.4: Workflow markers used in the manifests

The `publish()` method of a smart object is its capability of collecting in the manifest the rest of its capabilities and delivering it to the orchestrator in some way. `publish()` is an “implicit” capability that every smart object must implement in some manner and that represents the discovery phase. It creates and shares the manifest, for this reason it is not reported in the manifest.

4.3.4.1 Video Wall

The video wall (introduced in 4.2.1.1) has nine screens and a controller able to split a video content among them. Its class diagram, shown in Figure 4.10, is quite simple. It has a class that represents the video wall core and another that describes the screens. The first one inherits from `SmartObject`, while the second one from `Video`, which is an `Actuator`.

In this example, we supposed the video wall composed of screens that cannot work alone, without the controller, and we suppose it can only play videos. For this reason, **Screen** only inherits from **Video**. Multiple inheritance is also permitted, so it might also inherit from **Image** and, if the screens had a speaker, it would also inherit from **Audio**. Hence **Screen** is only able to start to play a video (**start(video)**) and to stop it.

VideoWall class gathers the nine screens in its **actuator[]** array, but it hides them, making it impossible for other external entities to use them separately. The other attribute (**tileConfiguration**) defines the current configuration of the video wall's tiles.

Figure 4.10: Video wall class diagram

VideoWall has six capabilities: the three inherited from the superclass, that every smart object must have, and three usable for the interaction phase that use the screens.

Methods **getId()** and **getPosition()** return the object's identifier and its location, while **publish()** share the object's module with the orchestrator.

configure(TileConfiguration) is the capability of change the tile's composition on the video wall. It is a getter method that modifies the **tileConfiguration** attribute. A change of this value will also change the physical configuration of the tiles. This capability may be called to configure the video wall off-line before the execution of a scene, but it may be also used as an action in a scene.

A video content is sent to a screen, or to a group of them, using **start(TileId, VideoContent)**. The parameter **tileId** identifies the tile on which the video must be shown, and the **videoContent** is the content to show. The video wall automatically checks the tile's configuration for **tileId** to obtain the screens that must be

used. Hereafter, it crop the `videoContent` in the right number of parts according to the tile configuration. Finally, it sends the cropped portions of the video to the correct screen. It is possible to call this capability several times to send a different content to each tile.

To stop a video, `VideoWall` provides the method `stop(VideoContent)`, which receives the identifier of the video to stop. This method could be also implemented using the tile identifier instead of the video to indicate what to stop playing, but for simplicity we only assume it works with the video identifier. Start and stop are two capabilities that come from the screens, but the video wall does not provide direct access to them. It has two capabilities that are wrappers of screens' start and stop: they add their own parameters, but to work they call their equivalent function of `Screen`.

Figure 4.11: Video wall manifest

Object's module contains the manifest and the driver. The manifest is obtained from the class diagram, considering the public methods exposed by a smart object and the public methods shown by its public sensors and actuators. In the Figure 4.2, the capabilities reported in the manifest were described using a natural language. Using the model, they are converted into methods. This conversion is showed in Figure 4.11.

The first one, give my id, for example, is converted into `getId()`. The d letter indicated that this capabilities may be used during discovery. All capabilities are converted following the same logic. As said before, `configure()` may be used during personalization and as an action during interaction phase. Method `publish()` is not included in the manifest because it represents this process of creation and sharing of the module.

The last line `Actuator::Video` is included for completeness. It says that the video wall uses one or more components (in this case actuators) that derive from `Video`. It also says the parameter `video` passed to `start(TileId, Video::VideoContent)` and `stop(Video::VideoContent)` comes from this class. This definition is useful for the recommender (see Section 5.4), which need to know the position in the model

of as many components as it can, in order to automatically adapt, configure and propose scenes to the users. If a manifest include a new class, reporting its chain of inheritance, it is possible to add it to module and then to use it as a known class.

4.3.4.2 Door

The smart door is an aggregation of two objects: the door itself and the smart lock. The door is the main object which controls the lock. The conversion of the door's scheme (Figure 4.3) into a class diagram is shown in Figure 4.12.

Door class has the three capabilities (`getId()`, `getPosition()` and `publish()`) and two attributes. The first one is `lock`, that is the reference to the smart lock object. The other is the magnetic sensor which is stored in the `sensor[]` array. Unlike the video wall's attributes, both the lock and the magnetic sensor are public, so their capabilities are directly accessible from the orchestrator, without the need of wrappers. To increase the flexibility of the model, `sensor[]` and `actuator[]` arrays admit an access modifier for each element, and thus it is possible to have a private sensor and a public sensor in the same vector.

The **MagneticSensor** class inherits from **DoorStateDetector** abstract class, which in turn derives from **Sensor**. **DoorStateDetector** sensors have a boolean field `isOpen` that describe if a door is open or closed. **MagneticSensor** implements the **Sensor** capabilities and it uses `isOpen` and `isSensing`. It has `startSensing()` and `stopSensing()` that enables and disable the sensor respectively.

`setEventCallback()` notifies when the door is opened or closed (when `isOpen` changes its state). `getValue()` allows the orchestrator to read `isOpen` field. The capability `isOpen()` is added by this particular magnetic sensor to permit to check the state of `isOpen`. These last two capabilities, in this case, working with boolean values, return the same value, but they operate in two different ways. `getValue()` directly returns the value of `isOpen`, working as a getter. `isOpen()` checks the value of `isOpen`, but it does not provide the attribute. If `isOpen` were a number that describes the opening percentage of the door, then `getValue()` would return a number, while `isOpen()` would still return a boolean value. They have been added to explain this difference and how they are included in the manifest.

The other smart object that composes the door is the lock. It has its own `getId()` and `getPosition()`, but since the smart object is seen as a single smart object, they returns the same values that **Door**'s ones does. To prevent this behaviour, they are not included in the manifest (ideally by **Door**'s `publish()`). On the other hand, **Lock**'s `publish()` collects all the **Lock**'s capabilities and reports them to **Door**. The last **Lock**'s capability is `configure(AccessList[])` which is used to set up a list of NFC tags that can open the door.

It has three fields: the `accessList[]` is private for security reason and accessible through the aforementioned method `configure()`; **NFCReader** and **LockMotor** are

Figure 4.12: Smart door class diagram

both public and they represent the lock's transducers.

NFCReader inherits from **Token**, which in turn extends **Sensor**. **Token** provides a **tokenId** field to the **NFCReader** that uses it to store the last read value. **NFCReader** only implements the three public capabilities defined in its parent **Sensor**.

The method **startSensing()** enables the reader, **stopSensing()** disables it and **setEventCallback()** notifies to the orchestrator every time a new tag is read.

The last concrete class is **LockMotor** that represents the latch bolt motor. It

derives from `ElectronicLock`, which in turn inherits from `Actuator`. `LockMotor` has only two capabilities: `start()`, that forces the latch bolt to move to the position stored in the inherited field `isLocked` and `control(IsLocked)` that permits the orchestrator to set up `isLocked`.

In this case, the manifest is larger than the one presented in Figure 4.3 due to the presence of sensors and actuators, that implement control capabilities such as `startSensing()` and `stopSensing()`. The smart door's manifest is presented in Figure 4.13.

As for the video wall, it is created gathering all the public capabilities from objects and components, respecting their visibility.

The two sensors have `startSensing()` and `stopSensing()`, which can be used during personalization to activate or deactivate the component before performing a scene, or during the execution of a scene as actions.

Figure 4.13: Smart door manifest

`MagneticSensor` has `isOpen()` and `getValue()`. The first one is used to check a condition (C), while the second one saves the door's state into a variable (V). These two methods, together with `setEventCallback()`, express the *Detect door state* capability. Therefore the smart door can notify when the door is open or closed, it can answer when the orchestrator asks it if the door is open, and finally it can share the door state with the objects that need it.

The electronic lock is reported in the last six capabilities. *Lock* and *unlock*, in this example, does not have a direct capability in the manifest. They are performed using `start()` and `control()`. Setting up an *open* state, calling `control()`, and then executing a `start()`, the door is unlocked, while setting up *close* and calling `start()` it will be locked.

`NFCReader` only has the `setEventCallback()` method to notify when a tag is read,

returning the value stored in the tag. The configuration of the access list is performed by the `Lock` class.

`getId()` and `getPosition()` are reported only one time using the `Door`'s implementation: the `Lock`'s version would return the same values, being part of the same object. Last three lines defines the relations between the classes in the model.

4.3.4.3 Drone

The last smart object modelled is the drone. As described in 4.2.1.3, it has a camera to take pictures, record video and send them to a remote node in real-time. In Figure 4.14 is represented the class diagram of the drone.

The camera is modelled with a concrete class that derive from `SmartObject` and that is controlled by the `Drone`. The components are three: the CCD sensor for the camera, and the GPS and the four rotors for the drone.

`Drone` allows the orchestrator to use the camera and the GPS, but it hides the rotors. The rotors are controlled by the drone, which automatically decides the speed for each rotor according to the direction and the inertial data provided by the IMU. The IMU component may have been included in the model, but, since its capabilities and state vector are not accessible, it does not bring any useful information to the orchestrator, it is not represented.

`Drone` exposes the three public methods of `SmartObject`: `getId()`, `getPosition()` and `publish()`. In this case, `getPosition()` is configured to return the position using the information provided by the GPS. This capability is thought to identify and locate the drone during the discovery phase: the GPS is able to provide the same or more accurate information about the position and it exposes its own capabilities to do that during the interaction phase.

The drone also has `move(Direction)` and `configure(Waypoint)`. The first capability allows to move the drone toward a direction in the space. The drone automatically decides the right speed for each rotator to produce the movement.

The second capability permits to set up a list of way-points providing the capability of automatic flight.

The camera is a smart object that has an imaging sensor and a network interface used to send the captured data. It has a CCD sensor whose capabilities are usable by the orchestrator. `Camera` implements the three public methods of `SmartObject`, but they provide the same values as those returned by `Drone`'s methods.

Moreover, it has a `send(Destination, Frame)` method. It is used to send the captured data (photo or video) to a remote device. Without this capability, data could only be directly sent to the orchestrator, not to other nodes. In this example, we supposed that the camera's network interface is connected to the drone, so data are sent passing through the drone's network interface.

Figure 4.14: Drone class diagram

CCD derives from `Imaging`, which in turn inherits from `Sensor`. It exposes `startSensing()` and `stopSensing()` to enable or disable the sensor and two methods to capture and return an image. `getValue()` takes only one frame (a picture), while `getContinuousData()` captures frames continuously, creating a video.

GPS inherits from `PreciseLocation`. It provides all the capabilities of `Sensor`.

It can enable and disable the sensor, it can return the three coordinates with `getValue()` and `getContinuousData()`, or it can generate an event when the drone enters or leaves a preconfigured space (e.g. within a certain distance from a way-point), using `setEventCallback()`.

Finally, `Rotor` class describes the four drone’s rotors. They are controlled only from the drone, which changes the speed of each of them using `control(Speed)` setter method. The current rotation speed of each rotor is stored in the `speed` field. This class also has `start()` and `stop()` methods to turn on and off the rotor.

The manifest presented in Figure 4.4 is described, using the public methods and fields defined in the model, as shown in Figure 4.15. It is created with the same logic of the two previous examples.

Figure 4.15: Drone manifest

It has the fourteen public capabilities explained above. The rotors are hidden, they are indirectly controlled using the action `move()`. The camera is able to stream video and send pictures “implicitly” to the orchestrator with `getContinuousData()` and `getValue()`, or to another device using `send()`. For completeness, `Rotor` is reported on the bottom line, even if it is not used in the rest of the manifest.

4.4 Summary

This chapter proposed a general model able to describe every smart object. It has been created beginning from the analysis of the internal structure of a generic object and converting its components, states and capabilities into concepts of the OOP. Analysing a smart device, we first create an intermediate model that characterizes

its components: processing unit, memory, network interface, sensors and actuators. This model is then converted into a class diagram, using and reinterpreting some feature of the OOP. The class diagram correlates the components and the capabilities that an object should provide, defining the manifest and thus helping to manufacture the smart device and its module.

In this chapter, the class diagram is used in a “static” way. It describes the components of a smart object, the relations among them and the capabilities that the object can provide to the orchestrator, but it is not used referring to the workflow phases.

In the next chapter, we will describe how to use the model during each step of the workflow. How to publish an object’s module, how to use the class diagram to adapt and propose scenes to the dwellers of a smart space, and how to use it to orchestrate the interactions among smart objects.

Chapter 5

An Architecture for Smart Spaces Using the OOP Smart Object Model

In Chapter 4, it was introduced a model for smart objects based on the OOP. At the beginning of the chapter, we introduced a table with the definition of the main OOP features and our reinterpretation of them to describe a smart space using its smart objects. In this chapter, we will use those concepts to describe each step of the workflow to apply the proposed model.

Firstly, we will introduce a general overview of the system architecture, assigning the blocks of the technology agnostic architecture (see Section 3.3) to the devices that belong to a smart space and that take part to the execution of a scene. Then, we will apply the smart object model to describe these actors and which of their components are involved in each phase of the workspace. Finally, in the smart space orchestration section, we will get into the details of the data exchanged during the interaction process.

5.1 Architecture Overview

As mentioned in the Section 3.3, only three kind of devices are necessary to cover all the workflow: a smartphone that acts as an orchestrator, a server that stores objects' modules and shared scenes for a space, and the smart objects that cooperate to perform the scenes. In the ideal situation, the components presented in the technology agnostic architecture are assigned as described in Figure 5.1.

All the logic is located in the user's smartphone. The server only stores the modules, for those smart objects that are not able to provide the automatic discovery, and the scenes shared by other users. Smart objects publish their module and are directly controlled by the orchestrator. Communications are made possible by a common bus, generally implemented by an IP local network, while less common communication

Figure 5.1: Architecture overview

technologies (for example IEEE802.15.4) are integrated by the use of gateways. The system workflow is also controlled by the orchestrator that employs the blocks from the bottom to the top (blue blocks for discovery, green for personalization and orange for interaction). Smart objects publish their modules and use their hardware components to perform their capabilities, controlled by the orchestrator.

Communications involve all the three actors, that can exchange modules, scenes, identifiers, locations and capabilities' data (data used for configuration and control during execution).

Figure 5.1 represents the ideal situation, where the user always has all the logic working in his smartphone. This solution has several advantages: in terms of privacy, user's data is not shared with other entities; in terms of efficiency, all the logic blocks may be implemented in a single application, increasing the speed of the system. On the other hand, some blocks may require an excessive computational power for a smartphone, resulting in excessive battery drain; the personalization blocks need the information stored in the two databases to work properly, which increases the number of network communications. In our implementation, we moved the two adaptation block marked with a dotted line to the server.

The behaviour of each block of the architecture can be described using the proposed model, analysing the capabilities, the relations between the components and the changes applied to the objects.

To do this, we first need to draw the model for the orchestrator and the server. In

this manner, we will have a model for every kind of actor of a smart space.

The orchestrator is a smartphone which may have physical sensors (accelerometers, gyroscope, microphone, camera, etc.), physical actuators (screen, vibrator, speakers, etc.) and virtual sensors and actuators represented by some of the applications installed on it. It can be represented with the model as a smart object, and also used as such during the interaction. Moreover, it has another part which contains the logic that allows it to control objects and scenes, manually and automatically during all stages of the workflow. Another important feature is its behaviour during the discovery. It already knows its components and capabilities and they are not thought to be used as external components by other orchestrators, so it does not have the `publish()` method. It does not even share its identifier and location because these values are used by itself.

The server is also modelled as a smart object that does not have any sensors and actuators. It is only able to store modules and scenes and to provide them to the orchestrator when required.

The behaviour of each logic block can be described using the model, reflecting the changes to the real objects.

5.1.1 Scene Description

In the previous chapter, we used a class diagram to describe the smart objects and their capabilities. That description was “static” because it treated the objects as independent entities, without defining interaction among devices and their cooperative behaviour. Now, we put together the models of various smart objects to represent a scene.

A scene is a composition of events, conditions and actions, created by the user or by an automatic system, and executed by the smart objects, under the control of the scene execution block. A scene, to be described, needs to contain the actors who take part in the process and what they are going to do. This kind of description is the same used in computer programming, where variables or instances of classes are first declared and then used in a logical sequence. In our system, we describe the scenes using the same technique.

Let us consider the following scene:

When I open the main room door, if it is between 9:00 and 12:00 of a working day, then play the demo video on the video wall using all screens.

This simple scene uses three objects and one capability from each of them. The objects are the door, the orchestrator and the video wall. The three capabilities are: to detect a change of the door’ state, to check current date and time and to play a video in a given configuration on the video wall. They are respectively an event, a condition and an action provided by the door, the smartphone and the video wall.

The scene can be written as a computer program with declaration and steps of an algorithm. Using pseudo-code it can be expressed as shown in the following program listing (5.1).

Listing 5.1: Pseudo-code of the scene

```

1 Door mainDoor = new Door("doorId");
2 Smartphone mobile = new Smartphone("smartphoneId");
3 VideoWall videoWall = new VideoWall("videoWallId");
4
5 ON
6   mainDoor.sensor[MagneticSensor].setEventCallback(isOpen == true);
7 IF
8   mobile.sensor[DateAndTime].checkTime(startTime = 9:00,
9                                           endTime = 12:00,
10                                          dayOfWeek = workingDay)
11 THEN
12   videoWall.start(video = demo.avi, configuration = "1:3x3");
13
14 (Door) Sensor::DoorStateDetector::MagneticSensor
15 (Smartphone) Sensor::Calendar::DateAndTime
16 (VideoWall) Actuator::Video::Screen

```

The first three lines declare the objects involved in the scene, with their identifiers passed as parameters in the constructors. Then the scene behaviour is described using event, condition and action and a keyword (ON, IF and THEN) that separates the groups of capabilities.

The cardinality of each group of capabilities is: 1 for events, 0..n for capabilities and 1..n for actions. Therefore, the simpler scene has only one event and one action. Between two conditions, it is possible to use only the AND connector, and it is possible to use the negation NOT on one condition or on groups of them. The set of functions {AND, NOT} is functionally complete, then it can express all possible truth tables. The execution of a scene is sequential. Methods are invoked on the object's component that provides that capabilities or on the object itself (videoWall.start()). The scene is executed by the orchestrator, which is responsible to receive the event and to dispatch each instruction to the right smart object.

Below the functional description of the scene, it is always reported the inheritance chains of the components involved in the scene. This part will be important for the portability of a scene because it permits to extend the object model including devices from other spaces. With a more complete model it will be easier to carry out the adaptation of a scene to a new space. The orchestrator completely ignores this section during the execution of the scene, it is only used to support the performing of the automatic personalization.

In this example, the last line is very important for the adaptation of the scene. Since the scene only uses a capability that directly belongs to the video wall object,

Figure 5.2: Scene representation with class diagram

it will be very difficult, for an automatic system that does not know the video wall's module, to determine which components act in the scene and what object can operate similarly to a video wall. The last line says that the video wall is composed of screens, and thus a screen could provide a reduced group of capabilities that work

similarly to the video wall's ones. Depending on the capability used in the scene, if a video wall is not available in a space, a screen may replace it, making the scene portable.

The class diagram can also be used to describe a scene, but, in this case, the order of the capabilities that belong to the same group is lost: it is still possible to determine the type of all the capabilities, the E-C-A order is immutable, but not the order between two or more capabilities of the same type (conditions or actions). On the other hand, with the class diagram it is possible to see on the same graph the relations among the objects which act in a scene, their components and the hierarchy of the capabilities. This information is very useful during the personalization, where scene are manipulated by users and automatic systems. In this phase, any additional data can help to create and to adapt scenes.

In the Figure 5.2 is shown the same scene described using a class diagram. Only the components that take part in the behaviour are represented. Multiple copies of `SmartObject` and `Sensor` classes are included, following the UML 2 internal structure notation (Miles and Hamilton, 2006, Chapter 11), to better illustrate the separation between the three involved objects.

In the real implementation, the code is saved in a XML file, but the all the system components treat it as an OO program. In the text we will use a Java like programming language to better illustrate the behaviour of the parts of the system with the description of a scene. The inheritance chain part will be often omitted in the next sections as unnecessary to describe a scene's behaviour.

In the XML, an additional field is also added: it is the description of the scene in a natural language description (see the scene proposed at the beginning of this subsection, *When I open the main room door, ...*). This text is automatically generated by combining together the phrases associated to each capability and stored in the modules. These descriptions are very useful for the users for both recognizing what a scene does without opening and understanding it, and when one of the automatic components proposes a scene to the user (e.g. see Figure 5.10).

A scene may have two optional labels, one contains a group name and the other is a boolean value that prevents or allow the use of the ECA behaviour to a recommender.

The first one permits to mark various scenes that belong to the same "group". This feature is useful for the user to better organize the scenes on the smartphone. Moreover, it allows to create groups of scenes that may work together to create a service (e.g. an ECA rule that switches on some devices when the user enters a room and that switches them off when the user leaves the room, these two scenes may be grouped under a "configure while I am inside" label). It may be used during the recommendation process to suggest a group of scenes that provide a service.

The second label determines whether a scene may be used by a recommender. It

permits to set an ECA rule as unmodifiable. If set, it will be never used in any recommendation process that uses a user's scene as an input. If a user manually shares a scene with the flag set, it will be maintained set in the copy in the repository. In this manner, it can be downloaded as it is. An example of a practical use of this field will be explained in the validation chapter (see Section 6.4).

5.2 Module Content

A scene is built combining the capabilities obtained from the objects' modules. A module contains all the information that the orchestrator needs to use the associated object. The main feature of the module is the list of capabilities. It is obtained from the model and it defines the steps of the module development process. Looking at the similarities with the programming languages, modules are dynamic libraries that promote the encapsulation and the reuse of the code.

A module includes a manifest, with the list of capabilities, the location and other information about the object, and a driver with the code that the orchestrator will use to invoke the capabilities on the object. Using the model, it is possible to obtain the basic capabilities that should be included in the manifest and in the driver. The driver contains compiled code, while the manifest gathers capabilities and other information. In the Chapter 4 we saw some examples of the lists of capabilities of the three smart objects analysed. In the list of capabilities are reported all the capabilities of the object, tagged with a marker that defines if each of them is an event, condition or action, if it can store a value in a variable and during which phase of the workflow it may be used. The list also contain the chain of inheritance of the object's classes obtained from the model.

Each capability has also a correspondent phrase in a natural language that describe what the capability does. This phrase is written so that combining capabilities to make a scene (e.g. *the door is opened* associated to `door.setEventCallback(isOpen == true)`), an automatic system can join the sentences from the capabilities, creating a description of the scene (e.g. *when the door is opened, then turn on the television*).

Apart from the list of capabilities, the manifest can contain the object's *identifier*, its *location*, a *configuration* and some *parameters* and *variables*.

The identifier is always present and it is the same value obtained during the discovery with the `getId()` method. It is used to associate the module with its object.

The location is optional in the module, but its value must be filled to consider a module ready to be used by the orchestrator. If the field does not specify a location, the user can assign it in the smartphone or the smartphone itself can assign a position according to its current location (e.g. a device discovered in the kitchen, with an empty location field in its module, will obtain the kitchen as its location,

Figure 5.3: Example of a module.

assigned automatically by the smartphone or manually by the user).

The configuration includes some specific fields that the orchestrator may need to be able to correctly interact with the object. This field is generic and may contain any kind of data. Example of data are the IP address of the object, the default amount of brightness of a bulb or a public key necessary to establish a secure connection with the object.

The variables are public value that a smart device can share with other devices through the orchestrator. They permit to create scenes where the behaviour of an object depends on the value provided by another object (e.g. *when the door of the fridge is open, then show me its internal temperature*, the internal temperature is read by a sensor, stored in a variable and passed to the smartphone, which will show it to the user). Variables have a public visibility in the orchestrator: all scenes can read and write variables and it is possible to read and write a variable without explicitly invoke a capability. In the code of a scene and in the module header, they are generally indicated with the \$ symbol.

Finally, parameters are additional data that can complements the capabilities. They are fields, written by the user, the orchestrator or an object during the execution of a scene, that provide additional information. Typical use of the parameters is to

preconfigure events or to obtain additional information during the fulfilment of an action.

The driver includes the code to perform the capabilities and the graphical interface that the orchestrator uses, during the personalization phase, to permit to the user to configure the object's behaviour. The inclusion of the graphic interface ensures that the module itself takes care of the configuration of the object, avoiding errors or potential dangerous situations. A very specific feature of an object will be configured only through its module, there is no other way to access to it and, of course, without the code contained in the driver it will be impossible to use the object, even if that feature has been included in a shared scene.

Modules are always packaged and compressed to reduce the need of storage space and the bandwidth for the transmission. In this manner, objects need less resource to store and send their module.

In the Figure 5.3 is shown an example of a generic module, with all the possible fields, and the fields filled with the values of the smart door described in the Subsection 4.3.4.2.

In general, the modules belong smart objects, but there are some of them that belong to the orchestrator itself. The latter are already present on the smartphone and they do not need to be discovered, they are always available. There are two kinds of orchestrator's module. The first kind gathers the blocks that permit to control the components of the smartphone. For instance, it includes a module to control the telephony, the accelerometer, the GPS, the vibrator, but also some software feature of the operating system such as the notifications. These blocks belong to a specific smartphone because they are hardware and/or operating system dependent. Third party application module also belong to this group, but they may be shared as a normal one. An example of them are blocks that permits to post or receive information from the social networks, retrieve weather forecast or read an email . They are associated to third party applications. In the model, their capabilities are described as part of the virtual sensors and actuators, because they do not interact with physical hardware.

The second kind is represented by a single module that permits to create chains of scenes. It is internal to the orchestrator application and it has four capabilities: two actions and one event. The actions are *activate a scene* and *deactivate a scene*, while the event permits to know when a scene has been activated by another scene: *when you are activated (by an activate scene action)*. When the first action is performed, it generates an event that can be captured by the event. The event is very particular, because it permits to an inactive ECA rule to capture the enable event. No other sensors can do that while there are inactive. These capabilities allow to connect different behaviours creating chains of scenes.

5.3 Discovery of Smart Objects

Creation and execution of a scene is impossible without a previous knowledge about the available object in the space and about what they are able to do. The first step of the workflow is thus devoted to give the orchestrator all this information. We identify two main type of discovery that a smart object can implement: automatic and manual.

The automatic discovery represents the best solution for the user, but it is harder to create and it requires more powerful hardware than the manual discovery. In this case, the object automatically sends its identifier, location and module to the orchestrator, without the user intervention.

In the ideal situation, the object stores its own module. The process is the following. It periodically broadcasts its identifier that is detected by the orchestrator, or it listens to a broadcast message from the smartphone. Since several smart objects are connected to A/C power or their power consumption is very low, the first solution is preferred. After this first packet, object and orchestrator establish a connection and the object's module is transferred, analysed and installed on the smartphone. This solution is hard to implement because it needs some storage space on the object, and a communication protocol, implemented both on the object and on the orchestrator, that defines the steps to reach the transfer of the module.

In order to reduce the hardware and software requirement of the smart device, it is possible to use a server which stores the modules of one or more objects. In this manner, the devices only need to broadcast their identifier. With this value, the orchestrator requests to the server the correspondent module. Even the identifier may be stored on the server, freeing up the objects from this task. In this case, discovering the server is enough: it will provide all the modules for all the devices available in a given smart space. On the other hand, the server needs a previous configuration and a copy of all modules that it will provide, in order to implement this solution.

The manual discovery requires the user intervention. He must interact with a smart object to begin the process. As described in the previous chapters, there are several ways to perform this interaction. Reading a code with the smartphone's camera or with the NFC reader, or manually interacting with the object to activate the publishing of its modules are some of them. In this case, there is not a periodical broadcasting of discovery messages, which make possible to identify a smart device at any time, but the sending of the first discovery packet is driven by the user *consciously*.

Discovery is the most object-dependent phase of the whole workflow. Each device may adopt the best solution according to its hardware. In our model, we represent this phase through the methods `getId()`, `getPosition()` and `publish()`, which

express the capabilities that a smart object has to provide its identifier, location and module respectively. The methods are included in the `SmartObject` abstract class. Therefore, according to the OOP, every smart device that inherits from `SmartObject` must provide its own implementation, using an automatic or manual solution. These methods represent a manner to provide information to the orchestrator. For them, we admit they can work in a non-canonical way (i.e. orchestrator calls them on a specific object). We only require that, when one of them is “invoked” by the orchestrator, it will obtain the related information in some way and in a reasonably short period of time.

In the architecture presented in Section 5.1, we included a modules repository in an external server. It allows to cover both discovery cases, storing the module if the object cannot do it, but leaving the possibility to provide the module to the devices that are able to.

The orchestrator has three blocks that work in this phase: the location manager, the resource catalogue and the resource filtering.

In the Figure 5.4 are represented the steps performed by the object, the resource catalogue and the location manager at the beginning of the discovery. All steps are

Figure 5.4: First steps of the discovery phase.

labelled with a number, which will be used to reference the description.

Location manager first receives the id of the object (1). In the case of automatic discovery, the smart device is able to send the identifier automatically to the location manager (typically through the Wi-Fi or Bluetooth interface), while in the case of

manual discovery, the user must somehow interact with the device (e.g. reading a QR code or a NFC tag). The identifier is passed to the resource catalogue. With this information, it can retrieve the object's module (2). The module is a compressed file and it is generally transmitted through a Wi-Fi or a Bluetooth channel. If the object does not have one of these technologies, an external server may provide the module for the smart device. Resource catalogue decompresses the module file and it analyses the content of the manifest (3). Additional data are sent to the location manager, while it stores the rest of the module that contains the capabilities and the driver (4), or it deletes these data if they are already available from an old module. Location manager controls the position of the orchestrator and of the objects in a smart space. It maintains a table that stores a tuple with at least *id*, *position* and *module reference* for each known object. The *position* value may be set manually or automatically by the orchestrator, if the object is not able to provide it. In the Figure, the module does not contain an hard-wired location, so the location manager assigns its current position to the module (5). Finally, the tuple is referenced to the module in the resource catalogue (6). The *id* identifies a unique device and it correlates a specific object with its entire module. Note that the identifier is always sent by the object during the discovery, before sending the module, and that the same value is also reported within the module itself. Using the first identifier it is possible to avoid the multiple download of the same module from the same object (i.e. when the user comes back to a previous space). Several identifiers may be correlated with the same driver and capabilities, in this manner, with only one copy of them it is possible to control more than one identical device. In a tuple, it may also be stored a specific configuration for a particular object. It is especially useful when a capability needs another device to be performed. For example, when a multimedia content is stored on a remote server on the local network, the IP address of the server depends on the space, and thus the same capability, from two identical televisions, works in the same manner, but it needs a different multimedia server. The user can delete these tuples to remove only some specific device. During deleting objects, the resource catalogue works as a garbage collector, removing the static part of a module when there is not any reference pointing to it. He can also directly delete the module, forcing the location manager to automatically remove all the tuples associated to it.

Resource filtering only passes to the personalization blocks the capabilities from the modules that belong to objects located in the same place of the user (default configuration). Thanks to this block, the user and the adaptation blocks will see only the objects available in that space. The filter may be disabled or configured to work in a different way, for example creating zones that include different smart spaces or parts of them, or adding specific groups of objects. This block does not work only on the modules, but it works also on the capabilities, providing to the personalization blocks events, conditions and actions, which will be the units handled during

personalization.

Figure 5.5 shows an example of collaboration among the three discovery blocks.

Figure 5.5: Discovery blocks working together to filter capabilities

Let us suppose that the user is initially in the kitchen. His smartphone discovers a television. Regardless of the discovery method, `getId()` and `getPosition()` return `48:2C:6A:1E:59:3F` and `kitchen` respectively. These values are stored in a tuple in the location manager. Then `publish()` retrieves the module for that television. Resource catalogue stores the module and the location manager links it to the tuple. The user goes to the living room and he discovers a sound system and a television, which is identical to the one in the kitchen. Two tuples are added in the location manager, while only the sound system's module is stored in the resource catalogue, since the television module is the same for both spaces. The correspondent television's tuple is referenced to the old module and the new one is deleted because it is a copy of the old one.

While the user is in the living room, he decides to personalize the space. When he opens the application and selects the personalization option, the resource filtering automatically processes the available information to show to the user only the capabilities usable in that space (default configuration).

Default configuration is passed to the location manager, which detects the current location and selects the right tuples. The correspondent modules are unpacked in the resource catalogue and their content sent to the resource filtering. The latter

sorts the information, add the object identifier and applies other filters, if any. The object identifier will be used to know on which particular object to invoke the capabilities. Then all data is sent to the personalization layer.

If the user goes back to the kitchen, `getId()` will return an already known identifier, and thus the television module will not be downloaded again. Moreover, if the default filter is not modified, the resource filtering will stop passing the previous data to the personalization layer, exchanging it for data related to the television available in kitchen. This change only affect to what the personalization layer can see (mainly objects' capabilities), it does not modify any scene.

5.4 Personalization of the Smart Space

Discovery phase finishes when the orchestrator knows all the capabilities of all the objects in its surroundings. At this point, personalization phase begins. As for the discovery phase, we distinguish two main type of personalization: manual and automatic.

Manual personalization is performed by the user without the help of any automatic tool, but the resource filtering and the scene compatibility analysis blocks. The first one can help the user to filter the available capabilities, while the other block analyses the manually created scenes to prepare the objects for the execution and, possibly, it can detect errors and inconsistencies in the scenes. In the workflow, the manual personalization is called scene creation. The choice of this name is due to the fact that it is the only manner to build a scene from scratch: the automatic methods included in this work only process and modify existing scenes, but they do not originate new ones from nothing.

Automatic personalization always proposes scenes to the user, so we group this process and its sub-processes under the recommendation block in the workflow. In this step, a software system elaborates existing scenes, models of smart objects and user preferences to produce a new scene, suitable for the user and for the smart space where he is. In the workflow and in the architecture, we call this process adaptation, because it adapts an existing scene to a new situation. The model of the smart object is used to determine what are the available capabilities in a space and how they can be employed to modify a scene. User preferences complete the personalization configuring the capabilities of the new modified scene for a specific person. Finally, the new scene is proposed to the user, who decides whether to enable or discard it.

Considering the role of the input data (scenes, model and preferences), we decide to adopt two different solutions to approach the problem. The first one is based on the reinterpretation of inheritance and polymorphism from the OOP and their application to the real world. The second one is a case-based reasoning system that

works when the first solution cannot provide a good result for the automatic personalization problem. Both methods are based on the proposed model. Preferences are additional data used to refine the final result. User preferences encompass a wide range of information and can include sensitive data, so we treat this data separately and always locally in the user's smartphone.

In the architecture (Figure 5.1), the *adaptation of my scenes* block may run locally or remotely. In our implementation, it is located on an external server. Moving the CBR into the same machine where the databases are located, the user's smartphone no longer needs to download all the existing scenes from the scenes repository to have the case base. In addition, since the modules repository probably knows more modules than a single orchestrator, the knowledge model available on the server is more complete and it is the same for every orchestrator.

The other block that belongs to the automatic personalization is the adaptation of previous scenes. This block analyses the previous scenes created for a specific space and, through statistical methods, it proposes a (previous) scene to the user, possibly refined with his preferences.

Every new scene, regardless of who created it, may be shared with other users. Prior to the sharing, sensitive data must be removed from the scene. All the modules know which of their variables and parameters may carry sensitive data. Before sharing a scene, each capability used in it is checked by the relative module, which deletes the fields that may contain sensitive data.

5.4.1 Manual Scene Creation

Manual scene creation is the basic way to personalize a smart space. When this method is used, no automatic system intervenes in the making process. Only resource filtering and scene compatibility analysis blocks may help the user to filter capabilities and to check the correctness of the final scene respectively. The manual scene creation block presents to the user the graphical interface contained in the modules to configure them and it checks the cardinality of events, conditions and actions during the creation. In a ECA rule there must always be one and only one event and at least one action. More actions and any number of conditions are admitted.

In the Figure 5.6 are shown the two main interfaces that the user employs to create a scene. In the first figure (5.6a), it is represented the general interface of the manual scene creation block. It uses yellow icons to identify events, green icons for conditions and blue icons for actions. The system automatically controls the cardinality of the capabilities (in the Figure, below the event, it is only possible to put a condition or an action). Icons are also included in the objects' modules, together with the graphical interface.

Figure 5.6: Example of scene creation

Each capability has its own GUI that permits to configure its parameters. In the second figure (5.6b), it is shown the graphical interface that belongs to the `setEvent-Callback()` capability of the door. This GUI permits to setup the clause that the door's sensor will check: it will trigger an event when it will be opened.

With the combination of these two interfaces, the user is able to create a scene. The manual creation block will automatically convert it into the code (e.g. Listing 5.1). The same interface can be used to modify existing scenes.

All scenes can be manually or automatically activated or deactivated. There are two special actions that permit to enable or disable a scene from within another scene or from the scene itself. With these two capabilities is possible to create chains of ECA behaviours (e.g. when a scene terminates, it enables another one) and scenes that can be executed only one time (e.g. when a scene ends, it is disabled by itself). The first case (chain of scenes) is useful to create hysteresis behaviour or to invert

the state of an actuator by repeating the same event, such as **scene1**: *when the temperature rises above 27 °C, then switch on the air conditioner, activate scene2 and deactivate scene1* and **scene2**: *when the temperature drops below 23 °C, then switch off the air conditioner, activate scene1 and deactivate scene2*.

The second one permits to create one-time-scenes that must be reactivated manually or by another scene, such as **scene3**: *when I open my office's door, power on my computer and deactivate scene3*: if I power off the computer at the end of the working day, the scene will not power it on again when I open the door to leave the office. Another ECA rule can reactivate the scene3 during the night for the following day.

A user may also share its scenes or download scenes shared by other users. This case falls into the manual creation because no automatic system modifies any ECA rule. Shared scenes are stored on an external repository and organized by spaces. Before uploading a scene, it is analysed and modified in order to remove sensitive data. Each module must know which of its fields may contain sensitive data and it is responsible for the deletion. Additional blocks on the smartphone or on the server may be added to supervise this process.

5.4.2 Recommendation

The other two blocks of the personalization phase that can modify a scene are grouped under the name of recommendation. They modify an existing ECA behaviour and propose it to the user, who may configure the smartphone to always accept and enable the scenes or to manually decide what to do.

The first block adapts the scenes that the user brings in his smartphone when he goes to a new space. Using the model, it is possible to reinterpret and to reflect to the real world the concept of polymorphism to exchange the objects that act in a given ECA rule. In the next subsection, this process will be explained.

The second block uses the scenes shared by other users who have been to the same space before. With a statistical analysis, it is possible to recommend to the user the most shared scenes, possibly perfected with his personal preferences.

Sometimes, the first block is unable to provide an adapted version of any of the user's scenes. It can be caused by the lack of similar smart devices from those used in the previous scenes. In this case, a case-based reasoner can use the shared scenes as the memory cases, the user's scenes as the target problems and the object model as the structural knowledge model to retrieve a similar scene and to suggest it to the user.

The CBR can solve the adaptation problem, but it needs scenes from both the orchestrator and the repository. Since the repository generally stores more scenes than the user's smartphone, putting the CBR in the same repository machine reduces the bandwidth and increases the general performances of the system.

5.4.3 Generalization and Specialization

The first block we introduced in the previous section tries to adapt the user's scenes to a new environment. The idea behind this block was inspired by the structure of the proposed model and by the behaviour of the OO compilers and the way they treat the subtype polymorphism.

The adaptation process is shown in the Figure 5.7. Generalization and specialization are described in detail in the next two subsections.

Figure 5.7: Adaptation of a scene through generalization and specialization

The adaptation block converts a scene in a generalized version of itself, using the model to obtain the inheritance tree and moving from the leafs (physical sensors and actuators) towards the root. The generalized version will contain only the necessary capabilities for performing the scene and the abstract classes to which they belong, losing all the information about the smart objects involved in the original version. Finally, the generalized scene is adapted to the new space looking for objects whose components derives from the abstract classes remained after the generalization process. The result may be refined using parameters and preferences that were present in the original scene, in other scenes or simply preconfigured in the user's smartphone.

Depending on the available objects in the new space, the new scene may be a reduced version of the original one and it may have a similar behaviour (not exactly the same). For this reason, at the end of the process, both scenes are saved, but in the new scene a field that stores a reference to the original one is added and it is marked as a copy. In this manner, if the user goes to a third space, only the original scene will be analysed because only this one has the full behaviour and it will generate the complete generalized version. If the specialization fails, the new

scene may be analysed to try to create an adapted version of this one. In general, for each new space, at most one adapted version for each original scene is created.

The adaptation requires that the block that performs generalization and specialization knows all the objects available in the new space, their inheritance chains contained in the modules and the inheritance chains of the components involved in the scene. The first two information can be easily obtained doing a discovery or manually installing all the modules on the device that will carry out the process. The last information is stored in each scene, below the ECA behaviour. The system will build its model from the gathered information, marking objects and components according to their location. Generalization and specialization will work on this model to operate the adaptation.

5.4.3.1 Generalization

Generalization is the first step of the adaptation process. It is performed using the smart object model. The model is a tree where the `SmartObject` class is the root and the physical sensors and actuators the leafs. The relation between `Sensor` class and `Actuator` class with `SmartObject` is an aggregation. Considering only the inheritance relations, in the tree there are two sub-trees where `Sensor` and `Actuator` are the roots. The generalization mainly works on these two sub-trees. The complete model is only used when there are capabilities that belong to a smart object instead of a component (e.g. in the Figure 5.2, the `start()` method that belong to the video wall).

For both the generalization and the specification, the names of the variable used in the algorithm follows the definition of Figure 5.8.

The basic idea of the generalization is to move on the model, converting the concrete components of a scene into their abstract parent classes. The Algorithm 5.1 shows the steps performed by this part of the adaptation process.

It analyses the behavioural section of the original scene. For each capability that it finds, it saves all the parameters. They will be possibly reused after the specialization to complete the adapted scene. If the capability comes from a component stored in `sensor` or `actuator` array of a smart object, then the parent class of the component is searched. This step is the most important of the entire generalization. Using the model, it corresponds to move from the concrete component class (a leaf) towards the root along the inheritance relations, stopping on the first abstract class. Since the scene contains the inheritance chains of its objects, this step can be performed (faster) on that list instead of on the entire the model. The abstract class obtained is added to the generalized scene, including the capability and the configuration parameters.

In the case of a complex capability that is directly controlled by the smart object instead of one of its components (e.g. `start()` in the Figure 5.2), only an identical

Algorithm 5.1 Generalization algorithm

```

function GENERALIZE( $s_{original}$ )
   $s_{generalized}$  = new empty scene
  for each capability  $x$  from object  $O \in$  scene  $s_{original}$  do
     $p_x \leftarrow$  all parameters of  $x$ 
    if  $x$  comes from sensor or actuator arrays then
       $C \leftarrow$  class that performs  $x_i$ 
       $\mathbb{C} \leftarrow$  the first abstract parent class of  $C$ 
       $s_{generalized} \leftarrow s_{generalized} + \mathbb{C}.x(p_x)$ 
    else
       $\mathbb{C}[] =$  new empty list of abstract classes
      for each object  $O \in s_{original}$  inheritance list do
         $C \leftarrow$  concrete class from the list
         $\mathbb{C}[] \leftarrow \mathbb{C}[] +$  the first abstract parent class of  $C$ 
      end for
       $s_{generalized} \leftarrow s_{generalized} + O.x(p_x) \mid \mathbb{C}[]$ 
    end if
  end for
  return  $s_{generalized}$ 
end function

```

Figure 5.8: Reference for the variable names used in the algorithms

or very similar device will be able to perform the same capability. Therefore the algorithm will report it as it is in the generalized scene, including its parameters. To increase the chance of adaptation, the (abstract) components of that object are also added, executing the same generalization process as before. This time it must be done on the inheritance chains contained in the original scene, because the behavioural section does not provide the information about the internal component of that object. In the inheritance chain section, the algorithm looks for the lines that contain the smart object that performs the capability, and it adds to the generalized scene the penultimate class of the line (the last one is the concrete component, the penultimate is the first abstract parent class). Knowing the components, it will be possible during the specialization to try to split the complex capability into simple capabilities that can be performed by components, creating a similar behaviour of the original capability. An example of this case is a television that plays a video with audio. That capability can be split and performed by a projector for the video and a speaker for the audio. The algorithm returns the generalized scene.

Applying this process on the scene proposed in the listing 5.1, the result will be the scene of the listing 5.2.

Listing 5.2: Pseudo-code of the generalized scene

```
1 ON
2   DoorStateDetector.setEventCallback(isOpen == true);
3 IF
4   Calendar.checkTime(startTime = 9:00,
5                       endTime = 12:00,
6                       dayOfWeek = workingDay)
7 THEN
8   VideoWall.start(video = demo.avi, configuration = "1:3x3")
9   | Video;
```

The first two capabilities are performed by components, so they are exchanged by their abstract version. The last one is a complex capability performed by the video wall. It is maintained (using the object's class `VideoWall`, not the instance) and the abstract components of the video wall are added to it. In this case, it has `Screens`, which are subclasses of the abstract `Video` class. The latter is included in the capability.

All the information about the specific smart objects that execute the scene are lost. The original scene

When I open the main room door, if it is between 9:00 and 12:00 of a working day, then play the demo video on the video wall using all screens.

becomes

When something (with a door state detector sensor) is opened, if it is between 9:00 and 12:00 of a working day (checked by some type of calendar), then play the demo video on a video wall using all screens.

Any smart device with a door state detector may trigger the first event, and any device able to check the day and time may check the condition. The action should be performed by a video wall, but probably some video device could do something similar.

The abstract classes used in the generalized scene are a sort of empty slots that the specialization will try to fill with the smart devices available in the new space.

5.4.3.2 Specialization

If the generalization exchanges the components moving on the model upwards, the specialization process does the opposite, moving downwards. The idea behind this step is inspired by the polymorphism in the OO programming.

One of the fundamental principles of OO design is the Liskov substitution principle (LSP). It was initially introduced in 1987 ([Liskov, 1987](#)) and formally formulated in 1994 ([Liskov and Wing, 1994](#)) as:

Subtype Requirement. Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .

It says that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. Applying the LSP to smart spaces, we can exchange smart objects, sensors and actuators, from a generalized scene, with other ones, respecting the relations described with the model, and obtaining the same behaviour.

The specialization operates the substitutions on a generalized scene using the polymorphism. The algorithm is reported below (Algorithm 5.2). For each capability

Algorithm 5.2 Specialization algorithm

```

function SPECIALIZE( $s_{generalized}$ )
     $s_{new}$  = new empty scene
    for each capability  $x$  from obj  $O \in$  scene  $s_{generalized}$  do  $\triangleright O$  may be abstract
         $p_x \leftarrow$  all parameters of  $x$ 
        if  $O$  is a child of Sensor or Actuator then  $\triangleright$  is abstract
            for each child  $C$  of  $O \in$  new space do
                if  $O.x = C.x$  &  $C.x$  accepts  $p_x$  then
                     $s_{new} \leftarrow s_{new} + C.x(p_x)$ 
                    break
                end if
            end for
             $D[] \leftarrow$  all children of  $O$ 
            SEARCH_SIMILAR_CAPABILITIES( $D, p_x, s_{new}$ )
        else if  $O \in$  new space then
            if  $O.x$  accepts  $p_x$  then
                 $s_{new} \leftarrow s_{new} + O.x(p_x)$ 
            else
                 $D[] \leftarrow$  all children classes of classes reported with  $O$  in  $s_{generalized}$ 
                SEARCH_SIMILAR_CAPABILITIES( $D, p_x, s_{new}$ )
            end if
        else
             $D[] \leftarrow$  all concrete classes  $\in$  new space
            SEARCH_SIMILAR_CAPABILITIES( $D, p_x, s_{new}$ )
        end if
    end for
    return  $s_{new}$ 
end function
    
```

in the generalized scene, it checks whether the component is a child of the **Sensor**

or **Actuator** class in the model. If so, it takes that child class and it looks for its the leafs. The leafs are the physical components, available in the new space, that probably are able to perform the capability. The algorithm checks whether the first found component accepts the parameters saved for that capability. If so, component, capability and parameter are included in the new scene and the process continues with the next capability. If not, the next component is considered for the acceptance test.

In the case of a complex capability, only an identical smart object can execute exactly the same behaviour. Therefore the algorithm looks for an identical device. If it is found, the acceptance test is done. This step is necessary because the new object may differ from the original one, and thus it is possible that they do not work with exactly the same data. If the test is passed, the new smart device is added to the scene, if not an alternative solution should be found to try to include the original capability in the new scene. The same happens with the “non-standard” capabilities that belong to a component, but that are not described in the model, and when there is not any identical object able to carry out a complex capability in the new space.

The search for an alternative solution is reported in the Algorithm 5.3. It decomposes complex capabilities and complex parameters into more simple pieces and then it combines them trying to recombine all the parts of the parameter. Most of times this algorithm is not able to find a solution. When it happens, the capability is not included in the final scene. When the entire specialization process ends, it

Algorithm 5.3 Search similar capabilities algorithm

```

function SEARCH_SIMILAR_CAPABILITIES(list_of_classes,  $p_x$ ,  $s_{new}$ )
  for each class  $E \in list\_of\_classes$  do
    for each parameter  $q \in p_x$  do
      if  $E.x$  accepts  $q$  then
         $s_{new} \leftarrow s_{new} + E.x(q)$ 
        remove  $q$  from  $p_x$ 
      end if
    end for
  end for
end function

```

returns a new scene. Before continuing with the adaptation task, the cardinality of events, conditions and actions is tested. If it does not have one event and at least one action, the adaptation is considered failed. If the test is passed, the values of the parameters may be revised and changed, for example considering the values used in other scenes shared by other users in the same space. In this case the scene is portable.

The new scene can be a reduced copy of the original one. Comparing the two scenes it is possible to estimate a similarity value. It may be used to restrict the final test instead of use only the cardinality. In any case, the cardinality test cannot be skipped and it represents the least restrictive filter that a scene must pass to be considered a scene.

5.4.4 Case-Based Reasoner

The adaptation process may fail due to several factors. The most common is the lack of smart devices able to provide the minimum number of capabilities necessary to create a valid scene. An additional block has been added to try to solve these cases. It is a CBR system. It takes a scene that was not possible to adapt to the new space, and it searches the most similar scene to that one from a repository of shared scene for the space.

CBR is a problem solving paradigm from artificial intelligence, built upon a rule of thumb suggesting that similar problems tend to have similar solutions ([Bergmann et al., 2009](#)). The idea of CBR is to exploit the experience from similar problems in the past and to adapt their successful solutions to the current situation. The process model of the CBR is a cycle (Figure 5.9) organized around the knowledge base. In a general CBR, the knowledge base contains a database of problems with their solution. Our implementation is slightly different. Our problem is always to find/generate a similar scene to a given one, and our database of cases is the scene repository associated to a space.

The CBR cycle consists of four steps: retrieve, reuse, revise and retain. They corresponds respectively to obtain a group of similar problems to the given one from the database; to create a new solution for the problem using the group of cases get before; to check if the new solution is suitable for the given problem; and to add the new solution and the new problem to the database of cases. In our implementation, we will retrieve only the most similar scene from the repository. The reuse will be the adaptation of the found scene with the parameters stored in the given one. The revision of the solution will be performed by the user who may accept, discard or modify the proposed scene. Finally the retain will add the scene obtained during the entire process to the repository of scenes.

Apart from the database of cases, a CBR needs a procedure to determine how similar are two cases. This procedure is called similarity function and it is formalized as a function $sim : P \times P \rightarrow [0..1]$, that compares two problems and returns a similarity assessment as a value from $[0..1]$, where 0 is totally different and 1 is equals. The function must be modelled as a part of the general knowledge because it operates on that specific cases and it must know how to determine if two cases are similar or not. The use of a taxonomy is possible to define the procedure and the distances between the cases that it will use ([Bergmann, 1998](#)). Our class model

Figure 5.9: The CBR cycle

represents a taxonomy, so it is perfect to create this level of knowledge for the procedure.

In addition to the model, we also considered the structure of a scene, the cardinality of the capabilities and the parameters to design the similarity function. Observing the cardinality permitted for events, conditions and actions, it is evident that the event is the most difficult capability to exchange, due to its fixed cardinality of one, and the actions are in the second place because at least one must be present in a scene. The value w is used to assign a different weight to each kind of capability. We use $w = 5$ for events, $w = 4$ for actions and $w = 3$ for conditions in our similarity function. Our general function that compares the input scene (I) with each scene stored in the repository (R) is:

$$sim(I, R) = \frac{\sum_{i=1}^n w_i f(c_i^I, c_i^R)}{\sum_{i=1}^n w_i} \quad (5.1)$$

where $f()$ is another similarity function that works on the capabilities instead of on scenes. This function compares the capabilities, using the model, and the parameters. It divides events, condition and actions and then it works separately on each

group. Since the number of conditions and actions may be different in I and R or it may have a different order in the two scenes, each combination between the two groups of conditions and actions must be checked, maintaining the pair(s) with the highest value and not repeating the same element in the pairs. The same happens with the parameters of the capabilities. In both cases, also repetitions are taken into account (the same capability may be used one or more times in a scene and a parameter may be used one or more times in a capability). The $f()$ calculates the maximum similarity values for each considered group by summing the best combination of elements from each of them. The function must solve the assignment problem:

$$\left\{ \begin{array}{l} \max \sum_{i=1}^m \sum_{j=1}^n s_{ij} x_{ij} \\ \sum_{i=1}^m x_{ij} = 1 \\ \sum_{j=1}^n x_{ij} = 1 \\ x_{ij} \in \{0,1\}, \quad i = 1, \dots, m \quad j = 1, \dots, n \end{array} \right. \quad (5.2)$$

where x_{ij} is a binary variable equal to one when the part (capability or parameter) i from scene I is assigned to the part j of scene R , it is zero otherwise. The value s_{ij} is the similarity between part i and j . It is different for capabilities and parameters, but in both cases it is estimated comparing the two elements and their relation according with the model. For capabilities it is:

$$(\text{capabilities}) \quad s_{ij} = \frac{1}{2}c_{ij} + \frac{1}{2}\widehat{f}(p_i, p_j) \quad (5.3)$$

the similarity value c_{ij} between the capabilities without considering the parameters and the normalized $\widehat{f}()$ function executed on the parameters, both part of the equation are weighted. The value c_{ij} is calculated using the model with a function similar to the specialization algorithm (Algorithm 5.2), but adapted to return a number instead of a scene. It returns a value between zero (c_i and c_j completely different) and one (c_i and c_j equals or totally compatibles).

The function $f()$ is called recursively on the parameter of each pair of capabilities considered by the external $f()$. For the parameters, the value s_{ij} is calculated comparing the type t of the parameters, not the content. The type of the parameter is defined the classes in the model. The formula used is:

$$(\text{parameters}) \quad s_{ij} = \begin{cases} 1 & \text{if } t_{c_i} = t_{c_j} \\ 0 & \text{if } t_{c_i} \neq t_{c_j} \end{cases} \quad (5.4)$$

The normalization of $\widehat{f}()$, when c_{ij} are processed, is done dividing the result of the function by the maximum cardinality of the parameters of c_i and c_j :

$$\max_{c_{ij}}(|p_i|, |p_j|) \quad (5.5)$$

It avoids cases where the parameters of a capability are a subset of the other (e.g. $c_1 = p_0p_1$ and $c_2 = p_0p_1p_2$). Without the normalization, the algorithm would calculate a similarity of 2 (p_0 and p_1 are exactly the same in both capabilities, so their similarity is 1 and the sum 2). This value is divided by the maximum number of element of both capabilities, in this case 3. In this manner the result is normalized (it is always scaled in the range $[0..1]$), and the lack of the element p_2 penalizes the final result of a one third factor. The final value will be 0.6.

Considering that the number of conditions, actions and parameters is generally low (three or four capabilities per scene and one or two parameters per capability), an algorithm that guarantees the optimal solution may be used to solve the assignment problem. We decided to use the Harold Kuhn's well-known optimized Hungarian algorithm (Kuhn, 1955), using the implementation modified for rectangular matrices (the number of elements in the two sets may be different). It has a $O(n^3)$ time complexity, but it guarantees the optimal solution. Moreover, scenes have a single event with only one parameter (the clause), then the algorithm will work on a matrix of only one element for both events and parameters of the events. If the length of the scenes grow, it may be substituted with a sub-optimal but faster heuristic algorithm.

Coming back to the $sim()$ function, the weights must be assigned to each type of capability (events, conditions and actions). Before that, each group must be normalized as is the case with the parameters. Each of them is divided by the maximum number of elements between the two groups considered. In this manner, the original definition $sim : P \times P \rightarrow [0..1]$ is maintained and the CBR engine can use it without adaptations. The function $f()$ works on each group separately and it returns the final similarity value for the entire groups. Therefore, the final formula will be:

$$sim(I, R) = \frac{w_e f(e^I, e^R) + w_c \frac{f(c^I, c^R)}{\max_c(|c^I|, |c^R|)} + w_a \frac{f(a^I, a^R)}{\max_a(|a^I|, |a^R|)}}{w_e + w_c + w_a} \quad (5.6)$$

where e , c and a are the groups of events, conditions and actions of the scenes I and R . The event is unique, so its value is already normalized by the function f .

Comparing a user's scene with all the scenes in the repository, the CBR obtains a list of its scenes sorted by similarity value. The most similar scene is selected, assigning to it the values of the parameters that were present in the user's scene. The new scene is suggested to the user, while the CBR waits for a response. If the

scene is accepted as it is or discarded, then the CBR discards it: in both cases it is already present in the repository. If it is modified by the user, when he finishes the modification process by saving the scene, this “new” scene is also sent back to the CBR and stored in the repository. Additionally, if the scene is discarded, the second most similar scene may be adapted and proposed, repeating the process an arbitrary number of times. By default, it is not repeated in order to maintain this process as less invasive as possible for the user.

It is possible to configure a limit for the similarity value. Only the scenes that reach that surpass that number will be considered for the suggestion. This limit prevents the recommendation of something that has nothing to do with the original user’s scene.

Apart from the revision step that is entirely performed by the user with a feedback, the other main difference of our system with a classic CBR is the base of cases. In a classic CBR, every new problem is stored with a solution and it increases the database size. In our implementation, the repository is mostly populated by scenes shared by the users, not by new problems.

5.4.5 Adaptation of previous scenes

Generalization and specialization use an algorithm based on the model to adapt a user’s scene. If the process fails, the CBR may try to find a similar scene from a repository. It is an hybrid solution that needs the user’s scene and those shared by other users and stored in the repository. The last recommendation block totally ignores the user’s scene: it only works on the repository.

It classifies the elements in the repository, using the similarity function described above, and it maintains a statistic of the most shared “class” of scenes. The similarity value is a number between zero and one. We decided to use a constant value of 0.7 as the similarity limit. Each time that a new scene is shared by a user, it is compared with all the others in the repository. If a similarity value greater than 0.7 is found, then the new scene is assigned to the same group of the one that is in the repository, and a counter associated to that group is incremented. If more than one comparisons returns a value above the limit, then the scene is assigned to the group with the highest similarity value. If none of the scenes reach the limit, a new group is created. The scenes also have a counter associated to each of them. If two scenes are exactly the same, only one of them is stored and the counter is incremented (also the group’s one).

When a new user enters the smart space, up to three (default value) options are proposed to him (the user can disable this feature or reduce the number of elements to see). The three scenes come from the three groups with the highest counters. For each of these groups, the one with the highest counter is selected. In case of a tie, the last shared element is chosen. The best solution for this case it is to check the

similarity of all possible pair of scenes within the same group, determining which one of them is the most similar to all the others, but the high number of elements in the repository makes it infeasible.

Figure 5.10: Example of three proposed scenes

Additional filters may be added by the user to request new shared scene, for example based on the position and on the smart objects near the user. One of the preconfigured filters requests new scenes each time that a new smart device is discovered. It works combining the location manager block on the orchestrator and the ability of the repository to search all the scenes that use a particular object and then order them by “popularity”. In the Figure 5.10 is shown an example of this filter in action. When the user discovers the weight scale, a request of the three most popular scenes that use that object is sent to the repository. It processes the request and sends back its three proposals. Finally, the user can decide what to keep and what to discard.

5.4.6 Scene Compatibility Analysis and Object Configuration

At this point of the workflow, the user has a scene ready to run. Just two operations remain to perform before the execution. Their role is to check the correctness of the scene and to prepare the smart objects for the performance.

The first operation is performed by a generic OWL-based rule engine. We added it to the orchestrator in one of our research work (Iglesias et al., 2012). It permits to set a group of rules that a scene or a group of scene must accomplish to be considered correct. It is not limited to analyse only one scene, but it can detect possible problems that may happen following the joint behaviour of a number of scenes. We set a group of four incompatibilities that the system can automatically check:

- *Exclusive resource incompatibility*: identifies those resources that cannot be employed at the same time in a scene (e.g., a user cannot be located in disjoint places at the same time or he cannot both receive and make a call);
- *Smart object interaction incompatibility*: identifies inter-resource incompatibilities, as smart objects interact among them given certain restrictions. For instance, it would not be possible to send some kind of data to a device that cannot use it (e.g. *ON event1 THEN send the video from the television to the lamp*);
- *Configuration incompatibility*: checks that each resource is configured according to its valid set of values (e.g., *ON event1 IF temperature < 55 °C THEN action1* would be a range-incompatible statement if the temperature sensor only range from -10 °C to +35 °C);
- *Functional incoherence*: prevents inconsistent actions to be configured (e.g., the same set of events and conditions cannot trigger opposite actions: *ON event1 IF condition1 THEN turn on the radio* rule is functionally incompatible with *ON event1 IF condition1 THEN turn off the radio*).

The second operation controls and prepares the smart objects for the execution. At the beginning, the object configuration was limited to check the parameters of all the events and to configure these values on the sensors. The role of this operation was to instruct the sensors to correctly trigger the correspondent event. This configuration is not an active part of a scene but it is necessary for the execution, so it must be done as soon as a scene is activated. Without this step a scene cannot be triggered and then work. Since it is possible to create chains of scenes where an action is to activate another scene, this process may be executed between two scenes and it must be very fast.

With the inclusion of the rule engine, we migrated the object configuration to this software component. It allowed us to improve the process that now not only configures the objects, but it is also able to detect problems performing this step, in particular:

- *Smart objects status*: a scene involving a particular smart object should be deactivated if the smart object is switched off, if it loses its communication capabilities, etc.;
- *Orchestrator communications status*: every scene requiring the mobile device to access a remote object should be deactivated if the mobile device loses communication coverage.

Moreover, since it must analyse the content of the scenes to find the parameters to configure for the sensors, it also analyses the rest of the capabilities. It can optimize the execution of the scene by exchanging the order of the conditions. ECA rules may involve evaluating ‘online’ conditions, i.e., those requiring to access an external resource. On the contrary, ‘offline’ conditions only need to access to parameters stored inside the user’s smartphone. So it is possible to configure scenes with both online and offline conditions.

Having information about the type of conditions and about the connector between them (possibly using De Morgan’s laws to convert AND to OR), it is possible to optimize the scene execution, trying to evaluate the minimum number of (only offline) conditions necessary to know the result of the whole condition sequence. Moreover, online conditions may be checked concurrently, increasing the performance, even when a lot of online conditions are present.

5.5 Smart Space Orchestration

After the discovery and personalization, a scene may be executed. The orchestrator starts listening. It waits for an event from one of the active sensors. When it receives an event, the execution begins.

The first step is to recognize the type of the event. All events are notified using the same packet. In the packet is contained the identifier of the event. With this technique it is possible to activate more than one scene sending only one packet. When the event is decoded, the value is passed simultaneously to all the active scene. The event code is received and processed concurrently by all the sensor’s modules of all the scene. Each module that recognizes the event code triggers its scene. Then is the turn of the conditions. Conditions need a bidirectional communication. The orchestrator asks a component, using the correspondent module, to evaluate a condition. The component analyses and checks the request and finally provides a

boolean answer. If more than one condition is needed, the orchestrator must wait to have the minimum number of answers to determine the value of the whole chain of conditions.

Finally the actions are performed. The command(s) that must be sent to the actuator are read from the scene and passed to the actuator's module. It sends the control signals to the smart object.

Every time that a module performs some operations, it is its driver who actually do it. The driver contains the compiled code that permits to the orchestrator to transparently interact with a smart object. The core of the orchestrator is a rule engine that reads the content of the scenes and communicates with the drivers to coordinate the correct execution. The engine works processing the scenes as a program. For each capability it finds in the scene description, it “invokes” that capability (method) on the smart object that exposes it. Thanks to the drivers, the engine does not need to know how to communicate with the objects and the capabilities are invoked as if they were remote procedure calls.

This is the canonical execution of a scene, but there is a piece among these elements that slightly changes the fulfilment of a scene. It is a variable: a value that can be used by the objects (read or written). Working with a variable requires an implicit interaction with an object. This interaction is not described in the scene, and thus not visible without looking in the configurations of each capability.

The value of a variable is obtained requesting the value to a component. The process is similar to a condition, but the returned value is not a boolean, it is a raw data. This data can be used as any other parameter of a capability.

In a scene description, a variable is directly indicated as a parameter in the capability that will use it. For instance, a scene that shows the temperature on a screen will include the variable temperature directly in the “show on the screen” capability. The temperature sensor is not even mentioned in the scene. The use of a variable may drastically change the real process of execution (and the bandwidth necessary) of a scene compared with its description. A scene that plays a real-time video captured by a drone on a television is described with a simple “play(video from the drone)”, but it actually need a lot of resource such as a real time communication for a video stream. Moreover, with the current implementation, all data must pass through the orchestrator, so the video stream will be received by the smartphone and forwarded to the television, needing even more resources. In the future implementation, this situations will be detected and possibly solved, for example by opening a direct communication channel between the drone and the television.

5.6 Summary

In this chapter, we have seen in details the architecture of the orchestrator and of the servers that help it to control the space. All its components and the data that it treats have been analysed. First, the mapping of the agnostic architecture on a real system have been proposed. Then, to better understand the data structure used in the architecture, we described the composition of a scene and of its building blocks: the smart object's modules. From that point, all the steps of the workflow have been analysed. One or more blocks from the architecture have been assigned to each step of the workflow, describing its specific tasks and internal structure.

All the parts of the system have been designed having in mind the smart object model. Each block works using the model. In the discovery, the capabilities to publish a module, the identifier and the position of an object are used. The personalization blocks employ the relations of inheritance to adapt or propose a scene to the user. The execution puts in relation the capabilities of different smart objects to produce a behaviour. A scene is described as an OO program and the system handles it like one.

In the Table 4.1, we proposed our reinterpretation of the main concepts of the OOP. The features related with the design have been explained in the previous chapter, with the definition of the model. The model draws a taxonomy of smart objects that classifies them according to their sensors and actuators. The orchestrator exploits the structure of the model to provide discovery, personalization and execution.

In the discovery, the encapsulation guarantees that the objects will only be used through their public capabilities. Therefore the orchestrator needs to know them. Otherwise it will not be able to use the object.

During the personalization, inheritance and polymorphism are the key features. Thanks to them, it is possible to create algorithms and similarity functions that use the model to look for similar objects. This point permits to exchange the devices that acts in a scene, but that are available only in a specific space, with others that are in a new space, without changing the general behaviour of the scene. It provides the portability of the scenes and it guides the automatic systems during the recommendation process.

The execution is done using messages and methods. Methods are the objects' capabilities. They are invoked in this phase to interact with the objects. Messages are the packets of data that orchestrator and smart devices exchange every time that a method is called.

In the next chapter, we will use some services, that we implemented in our laboratories, to validate the model and the architecture proposed. We will describe the objects that take part in the services with the model. The services will be

described as one or more scenes, as if it was a manual creation. Finally, we will analyse how the scenes are adapted when the user goes to a different space, where there are other objects.

Chapter 6

Service-Oriented Validation

This chapter presents the validation of the proposed model using some services, applying the model to their description and trying them in real environments. The inspiration for this kind of validation has been drawn from our previous works MECCANO ([Bernardos et al., 2013](#)), PERSEO ([Bergesio et al., 2015](#)), Follow-Me ([Follow-Me, 2015](#)) and the big display ([Bernardos et al., 2017](#)).

Beginning from these four works, we will prove the adequacy of the model to describe the behaviour of the smart devices and the interactions among them.

In MECCANO, we suggested a simplified version of the architecture proposed in this work and a mobile application. An enhanced version of that application will be our orchestrator for the validation of all services.

In the Chapter 4, we applied the model to three objects different from each other. That was a static description where only the devices and the capabilities were described. In this chapter, we will apply it to the interactions and the joint behaviour of the objects in a smart space.

The validation process will be the following. For each service, it will be presented a general view of the expected behaviour and which aspect of the model we want to validate. Next, the smart objects used in the service will be introduced. Finally, the model will be applied to the service for the validation. We will show how we can make use of the model and the architecture to control the execution of the services. The personalization is the most important step of the workflow and it is one of the open challenges of the smart spaces. For this reason, we will consider the use of the model in relation with the personalization step in all the four services.

6.1 Experimental Scenario

The real environment used for the tests is the Experience Lab located in the UPM's Montegancedo Campus. This laboratory is divided into eight zones equipped with

Figure 6.1: Three pictures of the Experience Lab.

smart devices that may be used to simulate different environments (e.g. office, home, hospital, hotel room, lounge, etc.). In the Figure 6.1 are shown three different areas of the laboratory. The space has also several systems that enhance the objects' capabilities: an indoor location infrastructure that permits to locate users and objects using Bluetooth low energy, a DLNA server that provides multimedia contents to the devices, a DLNA controller to handle contents and a smart home controller that centralizes data from multiple sensors and that controls actuators.

We developed a module for every available device that allows to the orchestrator to control the objects. Modules are stored on a local server and both manual and automatic discovery may be performed, using smartphone location or NFC tags to trigger the manual process or using UPnP for automatic discovery.

Another server stores shared and preconfigured scenes. It also includes a CBR system to recommend scenes to the users. The services proposed in this chapter will be tested in one or more zones of this smart space.

6.2 MECCANO

MECCANO was initially thought ([Bernardos et al., 2011](#)) to be a tool that permits to discovery smart objects and combines their capabilities to create automatic behaviours of the environment. Concepts discussed in that work was enhanced afterwards. In [Bernardos et al. \(2013\)](#), we improved the mobile application and we introduced the external servers to store and share modules and ECA rules.

MECCANO is not a service per se. It is a tool that supports the creation and the execution of services. In this first example, we will consider the creation of a very simple scene to validate some basic aspects of the model, such as the description of a service, the generalization and the specialization. The service will be tested in the hall of our lab (Figure 6.2) and in the meeting zone (Figure 6.1a).

The scene that we will consider comes from the following scenario. Let us suppose a user entering the hall of the lab through the main door. In that space there are

Figure 6.2: Hall of the lab with the smart door and the smart bulb.

a smart door (the main door) and a multi-colour smart bulb mounted on a work lamp. The user wants to automate the light to turn on and set the colour to green when the door is opened. He discovers the two objects with the smartphone and he uses the MECCANO mobile application to create a scene that represents that behaviour. The scene is:

*When the main door is opened,
then turn on the light
and set the colour to green.*

Then, let us suppose that the user goes to the meeting area. In this smart space there is another door (simulated with an Arduino and a magnetic sensor) and a wall spotlight (one of the five visible in the Figure 6.1a). The door is identical to the previous one, while the wall spotlight is different from the work lamp. It can be remotely switched on and off, but it has a monochromatic light.

The user wants to apply the same behaviour of the previous space. The mobile application automatically adapts the scene to the new space.

6.2.1 Smart Objects

The scene we propose is very simple and it uses the smart door described in the Chapter 4, a work lamp with a multi-colour smart bulb (Philips Hue LED light) and a wall spotlight with a monochromatic bulb in the second space. Using the model we can draw the class diagram of the three objects. The model of the door is reported in the Figure 6.3, the model of the work lamp is shown in the Figure 6.4 and the model of the wall spotlight is represented in the Figure 6.5. The door available in

the lab does not have an NFC reader and the smart lock as in the example reported in the Chapter 4, it only has a magnetic sensor.

Figure 6.3: Model of the smart door used in MECCANO service.

Figure 6.4: Model of the smart work lamp used in MECCANO service.

Figure 6.5: Model of the smart wall spotlight used in MECCANO service.

Following the description of the objects done with the model, we developed a module for each of them. The public methods represented in the grey classes are the capabilities that the objects have. Only those capabilities are included in the modules. The Philips Hue installed in the work lamp can modify its colour and brightness. In this service we will only change the colour of the light, so we model the LED light as if it were not able to control the brightness. This example shows how to hide protected methods that a sensor should have, according with the model,

but it does not have in the real world. Applying the OOP, these methods are left unimplemented in the module and they return an error if invoked. The same happens with the wall spotlight whose bulb can only be switched on and off. All the other inherited capabilities are left unimplemented in the module.

6.2.2 Service Discussion

In this service, we will validate the model applying it to the discovery of the objects and the creation and the adaptation of the scene.

6.2.2.1 Discovery

In the model, every smart object has three methods that intervene during the discovery: `getId()`, `getPosition()` and `publish()`. They provide the identifier of an object, its position and its module respectively. The implementation of these methods not only depends on the objects, but also on other additional services available in the environment (e.g. location system, module repository, etc.). In our lab, the doors and the wall spotlights are “smartified” with an Arduino Fio ([Arduino, 2010](#)), a small electronic board with reduced computational power. To show both manual and automatic discovery, the main door is equipped with a NFC tag and the simulated door and the wall spotlight have a Bluetooth proximity beacon. They provide a manual discovery. The work lamp includes all the hardware to provide the information needed during the discovery, so it can perform an automatic discovery.

Considering the main door, the user sees NFC tag and he reads it with the smartphone. The tag contains a special URL that is recognized by the orchestrator application. From that URL, the smartphone obtains the identifier of the door and its module. The location of the door is manually assigned by the user (the location manager in combination with the location service of the lab may be used for this purpose, but let us suppose that the user wants to set it manually). Reading the NFC tag, `getId()` and `publish()` are manually invoked, while `getPosition()` is not explicitly called, but its supposed return value is configured by the user when the module is downloaded.

The Philips Hue LED light of the work lamp supports the UPnP protocol. When the smartphone is connected to the same Wi-Fi network of the bulb, the latter sends its identifier, position and URL of the module through the SSDP protocol (discovery protocol used by the UPnP) to the orchestrator. The orchestrator stores all the information and it downloads the module from the URL. In this case, the three methods are invoked “automatically”: when the smartphone receives the SSDP discovery packet from the work lamp, it receives the identifier, position and URL of the module without explicitly calling the methods.

The simulated door in the meeting area uses exactly the same hardware of the main door, but it is not mounted on a real door. This time, instead of an NFC tag, the door has a Bluetooth proximity beacon that broadcasts its identifier and URL using the Eddystone-URL protocol (Google, 2015). When the user is sufficiently close to the door, the orchestrator receives the data from the beacon. This is a manual discovery process because the user must approach the door to obtain the data. The wall spotlight uses the same mechanism. The URL includes the identifier of the object (`getId()`) and the link to download the module (`publish()`) and to obtain the preconfigured position from a location database (`getPosition()`).

In all the cases, the three methods represented in the model are “invoked”. The important part of the discovery is the result: the fact that the smartphone knows identifier, position and module of all the objects at the end of the discovery.

6.2.2.2 Scene Creation

The discovery finishes when all the modules are installed and the capabilities contained in are ready to be combined to create a scene. Then, the user manually creates the scene. Some components of the architecture assist the user during the creation, filtering objects and controlling the cardinality of the capabilities. For example, the orchestrator helps him to search the first event applying the default location-based filter. Then, it only shows conditions and actions to set up the second capability (only an event is possible and it is already present). Finally, it only permits to add other actions (after an action only other actions are allowed). The user configures the scene (Listing 6.1) using the orchestrator’s GUI to select the capabilities and the module’s GUI to set up the internal configuration of each capability. Using the preconfigured sentences stored in the modules, the orchestrator also creates a description very similar to the scene described at the beginning of this section.

Using the name of the capabilities included in the model, the scene can be represented with the following pseudo-code.

Listing 6.1: Pseudo-code of the scene

```
1 SmartDoor mainDoor = new SmartDoor("mainDoor");
2 WorkLamp smartWLamp = new WorkLamp("light01");
3
4 ON
5   mainDoor.sensor[MagneticSensor].setEventCallback(isOpen == true);
6 THEN
7   smartWLamp.actuator[LEDLight].start();
8   smartWLamp.actuator[LEDLight].control(colour = "green");
```

Once the manual creation process is over, the scene is activated. The scene compatibility analysis is passed without issues and the door is configured to notify the

opening event. At this point the ECA rule is working. When the door is opened, the work lamp turns on and it sets its colour to green. For completeness, the class diagram of the scene, according with the model, is shown in the Figure 6.6.

Figure 6.6: Scene representation with class diagram

Scene creation shows that the model provides a reasonable representation of a scene putting in relation different objects. The class diagram statically describes all the components involved in an ECA rule, while with the pseudo-code it is possible to represent the order of the capabilities that compose the scene.

6.2.2.3 Adaptation

One of the open challenges of the personalization is the portability: the ability to move something that the user creates according to his preferences to another space and that it could work, at least in a similar way. Applying the inheritance, the generalization and the specialization of the OOP to the model, we can adapt a scene to work with the smart devices available in a new space, solving the problem of the portability.

The adaptation begins when the user goes to the meeting area. The adaptation may be manually started, but if the automatic adaptation option is selected, every time that the location system detects that the smartphone is in a new zone and every time that a new object is discovered, the adaptation process is started. We supposed that both smart objects in the meeting area need a manual discovery. Therefore, they are unknown when the location manager detects a change, and the

adaptation does not have any new object to work on. When the door is discovered, the adaptation is performed, but it cannot find an object in the new space to replace the work lamp, so it does not return any results. When also the wall spotlight is discovered, the adaptation is performed another time and now it has all the elements to generate a new scene. When a scene is adapted and proposed to the user, but not yet accepted and a new device that better adapt the new ECA rule is found, then the scene is automatically re-adapted and showed to the user, replacing the old one. In general, for each ECA behaviour available on the orchestrator, the automatic system tries to provide only one adapted scene, which behaviour is as similar as possible to the original one. It avoids to generate too much ECA rules that only confuse the user.

The generalization step is first performed. It uses the model to substitute the com-

Figure 6.7: Generalization of the proposed scene

ponents that provide the capabilities used in the scene with their abstract parent classes. Using the scene’s model, the process is shown in the Figure 6.7. In this case, no complex capabilities from objects or components are used, so also the references to the original smart objects are lost in the process. The generalization converts the original Listing 6.1 in the new Listing 6.2. This scene uses a generic smart object as the entity that provides all the three capabilities. The actual sensor and actuator are substituted with their parent classes `DoorStateDetector` and `Lightning`. The

scene is completely described in terms of abstract sensors and actuators. In a human language, it is something like:

*When a door state detector recognizes an opening,
then switch on a light
and set the colour to green.*

It loses importance to know who actually perform a capability: a car able to detect when the trunk is opened and with a multicolour light can perform the scene. A door with an embedded LED strip or a window, a smartphone's flash and a multicolour bed light can do the same.

Listing 6.2: Pseudo-code of the generalized scene

```

1 ON
2   SmartObject.sensor[DoorStateDetector].setEventCallback(isOpen == true);
3 THEN
4   SmartObject.actuator[Lighting].start();
5   SmartObject.actuator[Lighting].control(colour = "green");

```

After the generalization, the orchestrator performs the specialization step. Now it works on the generalized scene. It replace the abstract classes with their children: the physical component available in the new space. The orchestrator knows the model with the objects from the new space. For each abstract class it substitute the new component and the related object for the old class. Using the model, the process is shown in the Figure 6.8.

Figure 6.8: Adaptation of the general scene

In the new space, we supposed that the wall spotlight is monochromatic. Therefore, the last capability cannot be replaced. It is removed and the orchestrator tries to run the acceptance test. The acceptance test only needs that the resulting scene has one event and one action. The new scene accomplishes the requirement, so it is accepted. With the door and the wall spotlight, it is possible to create a similar scene to the original one. The code of the final scene is thus:

Listing 6.3: Pseudo-code of the adapted scene

```

1 SmartDoor meetingDoor = new SmartDoor("meeting_area_door");
2 WallSpotlight wallSpotlight = new FloorLamp("light02");
3
4 ON
5   meetingDoor.sensor[MagneticSensor].setEventCallback(isOpen == true);
6 THEN
7   wallSpotlight.actuator[Bulb].start();

```

Since the only capability that contained a parameter has been deleted, no refining of the parameters is possible in this example.

The specialization also generates the description in human language:

*When the door of the meeting area is opened,
then switch on the wall spotlight.*

This description is the text that the user sees on the smartphone when the process finishes. It gives to the user a general idea about the scene that helps him to decide without opening the ECA rule and seeing the details. He can accept or discard the adapted scene as it is or he can enter in the configuration and change it manually. Once accepted/modified, the scene is ready to work.

The model plays a very important role in the adaptation. The generalization and the specialization process with the properties of the OOP inheritance guarantee that objects can be replaced with others without altering the general behaviour of a scene.

6.3 PERSEO

PERSEO (Bergesio et al., 2013) was initially thought to be a way of managing multimedia between the available and most suitable interfaces. The concept was afterwards extended to non-multimedia content (Bergesio et al., 2015) and MEC-CANO mobile application was included as the configuration tool. The idea behind PERSEO is to “send” the mobile application that is running (in foreground) on the user’s smartphone to another device. Applications can be sent to another device for sharing purposes, for example to see a video on a television with other people, or to use that application on a more comfortable device, for example passing a web browser from the smartphone to a tablet or pc. In our previous works, we proposed

to send video contents to a television, audio to a sound system, photos to a digital photo frame, phone calls to a Bluetooth microphone/speaker and web pages to a tablet. The service does not transfer the applications, but only their contents to another device (to a similar or identical “application”). The transfer is started by the user, putting his smartphone next to/on/beyond a “trigger” object. The general form of a PERSEO service, described as an ECA scene, is:

*When the smartphone enters in the ‘influence area’ of an object
if the foreground application is ‘a given one’ ,
then the predefined action is launched.*

In the same manner, the service can be inverted when the user picks up the smartphone, returning back the content from the device to the mobile.

The service is provided in the form of several different scenes, marked to belong to the same group. All scenes begins with the same event. Then, each of them has a different foreground application and a different target object.

The service will be tested in our lab, in the meeting area (Figure 6.1a) and in the cinema area (Figure 6.1c). The event that triggers all the scenes is an NFC tag. In the case of the meeting area, it is embedded in the arm of one of the sofa. Putting the smartphone on the arm of the sofa, it reads the tag and triggers the scenes. The workflow of the service can be represented with the graph of the Figure 6.11. The

Figure 6.9: PERSEO workflow.

service is composed of five different scenes, one for each possible foreground activity

(audio, video, photo, browser or phone call). They may be manually created, as in the previous service, or downloaded as a service bundle from a repository.

Since the five scenes are quite similar, we will only analyse one of them. It will be the most complex one: the scene that handles the video contents. Adapting the general form, the scene for the video in the meeting area is:

*When the smartphone reads the sofa NFC tag,
if the foreground application is playing a video,
then stop the video on the smartphone
and play the (same) video on the television.*

To test the portability, the scene will be adapted to work in the cinema area, where a projector and an Hi-Fi system are available.

For the validation, in this second service, we want to prove that the model is suitable to describe a more complex scene than the previous ones. It will be also used to include applications as “virtual” components in a scene and to cover the communications between capabilities of different objects through the variables. The orchestrator itself will be also described. The model will be also used for the portability, but this time the objects in the second space will be quite different from the objects available in the first space.

6.3.1 Smart Objects

The main object of this service is the user’s smartphone itself. It is responsible to read the NFC tag, to execute the video application, to check the foreground application and to control the DLNA video stream sent to the television. In the top part of the Figure 6.10, the model is used to describe the smartphone. The video player application is modelled as a virtual smart object and it is part of the smartphone. The video player uses the speaker and the screen to play a video, but it does not provide access to them to external objects (private modifier of the actuator array in the figure). They are part of the smartphone, which can provide remote access to them. The activity manager is a virtual sensor. It is a part of the operating system of the smartphone able to control the applications that are running in foreground and in background. Both the `VideoApp` and the `ActivityManager` are virtual “things”. They are modelled in a different way because the video player is an applications that can run on any smartphone, so it has a module with the capabilities associated with the application. The `ActivityManager` is part of the operating system: its capabilities must be included in the module of the smartphone (or directly included in the orchestrator), its implementation strictly depends on the smartphone and on the operating system.

The other smart object involved in the scene is a television. The bottom part of the Figure 6.10 shows the model of this device (considering only the components

and capabilities needed to execute the service).

In the cinema area there are a projector able to play a video, but without a speaker and an Hi-Fi system. Their models are very simple and can be obtained by splitting the video and the audio components from the model of the television.

The transmission of the video stream is done with the support of a DLNA controller. However, it does not take an active role in the scene. The video player's module and the television's module use the DLNA controller, but it is transparent to the orchestrator and to the user. In the lab, the controller is implemented on a standalone machine, but it could be directly included in the video player application or in another application on the smartphone. Since the DLNA controller does not take an active role in the scene, it is not included in the model. It could be included, using the encapsulation, as a component of the video player and of the television.

Regarding the validation, in this subsection it is shown how the model can be used to describe a complex object such as the smartphone that includes the model of an application (video player) and of a part of the operating system that acts as a virtual sensor. In this case, the model not only shows the capabilities that must be included in the modules, but also their distribution in different modules. Each smart object must have its own module.

6.3.2 Service Discussion

6.3.2.1 Discovery

In this service, the workflow is slightly different from the canonical one. The first thing that the smartphone receives is a group of scenes instead of an identifier or a URL. The scenes include the references to all the objects and their modules needed to work. With this information, an “automatic” discovery is performed: the orchestrator automatically downloads identifiers, positions and modules for all the objects from a remote module repository. If some modules are not available, it is notified to the user, who can manually try to perform a manual discovery of the relative object.

The discovery part of the model is still valid: before the execution of the scene, the results of the three methods (`getId()`, `getPosition()` and `publish()`) must be available on the orchestrator.

6.3.2.2 Scene Analysis

The capabilities involved in the considered scene are four: the reading of a NFC tag, the detection of the foreground application on the smartphone, the interruption of the video on the smartphone and its resuming on the television. These capabilities are performed by the smartphone's NFC reader, by a software of the mobile's

operating system, by a video player application and by the television respectively. Moreover, it is necessary that the player application is able to provide its content to the television, possibly including the last played position to resume the playback from the same point. In our real implementation, the player application and all the multimedia devices available in the laboratory support the DLNA protocol, so a communication among them is always possible (e.g. to play a video on another device/application or to obtain the state of a device/application). The sharing of contents between capabilities is modelled and implemented using a variable.

Using the pseudo-code, the scene can be described in the following manner (Listing 6.4).

Listing 6.4: Pseudo-code of PERSEO video scene

```

1 Smartphone smartphone = new Smartphone(this);
2 Television tv = new Television("LivingRoomTV");
3
4 ON
5   smartphone.sensor[NFCReader].setEventCallback(token == "SofaID");
6 IF
7   smartphone.sensor[ActivityManager].isInForeground(VideoApp)
8 THEN
9   smartphone.VideoApp.stop();
10  tv.start($smartphone.VideoApp.videoContent);

```

The player maintains the state of its video after the stopping, so it is not necessary to save it before that action. The variable in the last action will retrieve that data before playing the video on the television. The model of the objects involved in the video scene is represented in the Figure 6.10.

Regarding the model, unlike the previous service, this time the scene uses some complex capabilities. They are used to control the video playback. Both the video player application and the television have a `start()` capability that internally coordinates the playback of the video on the screen and the audio on the speakers. Moreover, the activity manager uses a capability to check the foreground application that is not included in any abstract class of the model.

The model permits to include the complex capabilities and to describe scenes where a communication between capabilities is needed.

6.3.2.3 Adaptation

After trying the scene in the meeting area, the user moves to the cinema area. The adaptation block contained in the orchestrator tries to adapt the scene to the new smart devices.

The scene reports, after the code, the inheritance chains of the three objects that have a complex capability. The activity manager has a `Sensor::ForegroundApp` chain, while the television and the video player both have `Actuator::Video` and `Actuator::Audio` associated with their `start()` and `stop()` capabilities used in

Figure 6.10: PERSEO video service.

the scene. This information is used to search similar objects that could do the same thing, if the identical objects are not available in a space. Looking at the scene and at the model of the Figure 6.10, it is not so difficult to adapt. In fact, four of the components and one of the complex capabilities belong to the orchestrator. They are all local parts of the smartphone, so they are always the same in every smart space and they do not need any adaptation. The only two parts that need to be replaced are the television and the NFC tag identifier: the object that receives the last action and the parameter that is used in the event to trigger the scene.

The first object was a television in the meeting area. Therefore, any television or device able to play a DLNA streaming video is admitted for the substitution. Since both **Audio** and **Video** are reported in the scene, associated with the **play()** capability, it is possible to split the video into two multimedia streams and send them to two different devices. If in the new smart space there is a projector without loudspeaker and an Hi-Fi system, both of them DLNA capable, then the video may be sent to the projector and the audio to the Hi-Fi. The resultant scene will be very similar to the original one, although not exactly the same. Moreover, the separation

Figure 6.11: Generalized video service.

of the streams implies that the smartphone is able to generate them, starting from a video, and to send them to the objects in a synchronized way. Our video player module can do that, so it is not a problem for our system, but in other situations, having a similar scene it can mean having only audio or video playing, instead of both of them at the same time.

The second thing to replace is the identifier of the NFC tag. It is very hard to exchange because it is not a smart object as the others. It does not have a module, nor is it described in the model. Hence, there is not a discovery process for it. Some automatic system should alert the user that there is a NFC tag in the space. However, the only manner that a system may have to be aware of the presence of that identifier is by acquiring it from a shared scene. With a shared scene, it is possible to read that value, but it implies the possibility of using other adaptation methods (e.g. a CBR system). Without a shared scene, the user must manually read the identifier during the configuration to put the value in the new ECA rule.

In our lab, the tag is visible so, using the adaptation process, the user will manually modify the parameter after the specialization. He will open the configuration of the scene, then the setup GUI of the event, and finally he will change the value reading the new NFC tag. The resulting scene will be the same of the Listing 6.4 (admitting that there is a television in the new space), but with the identifier of the television and of the NFC tag replaced by the new values.

Using shared scenes it is also possible to get the NFC identifier, but having access to a scene repository, the employ of the recommendation blocks is preferred. In

this case, the probability of proposing the scene that the user expects is high, if the shared ECA rules are similar to the user's one.

Just for completeness, the code of the adapted scene with two streams is reported below (Listing 6.5).

Listing 6.5: Pseudo-code of PERSEO video scene

```
1 Smartphone smartphone = new Smartphone(this);
2 Projector projector = new Projector("CinemaRoomProjector");
3 HiFi hifi = new HiFi("CinemaRoomSoundSys");
4
5 ON
6     smartphone.sensor[NFCReader].setEventCallback(token == "CinemaID");
7 IF
8     smartphone.sensor[ActivityManager].isInForeground(VideoApp)
9 THEN
10     smartphone.VideoApp.stop();
11     projector.actuator[Video].start($smartphone.VideoApp.videoContent);
12     hifi.actuator[Audio].start($smartphone.VideoApp.videoContent);
```

The scenes for the other cases of the service are very similar. The only difference is the foreground application and the destination object. The restoring scenes uses the same objects, but they replace the NFC event by an accelerometer event: when the smartphone is moved, it activates one of the scenes is the second group. The sequence of the actions is inverted. The content on the external device is stopped and then restored on the smartphone.

Considering the validation of the model, it still provide an adapted scene, but this time the user intervention. In the second space it is required to adapt a parameter of a capability, not an object. The model cannot do it because it is oriented to objects and it only works with this entities. The solution is to model the NFC tag as an object, but in this manner the tag needs a module that contains its capabilities. The implementation is possible, but to simplify the system, we did not follow this way for the NFC tags available in the lab.

6.4 Follow-Me

Follow-me is a service that combines location capabilities with management of interfaces and video. The idea behind the service it that a user could move through different spaces while a video follows him, being always played on the video interface closer to the user. For a practical example, let us suppose a user that is watching a film on the television in the living room. Then, at a certain point, he decides to go to the kitchen to prepare some popcorn. With this service, the film will automatically stop playing on the living room television, and it will continue on the kitchen television. The process will work in any room where a video interface is available. The user can go back to the living room or go to the bedroom and finish to see the

(a) User watching a video in the relaxing area.

(b) User in the hall with the video moved to that area.

Figure 6.12: Follow-Me service.

film there, before sleeping, on a tablet, for example.

The particularity of this service is its ability to work seamlessly when the user goes to another space. Moreover, it stops working in the previous space.

Although the service works on any area of the lab, to simplify the analysis, we will consider the movement of the user through the relax area (Figure 6.1b) and the hall (Figure 6.2). In the Figure 6.12, there are two pictures of the service working. The user is moving to another zone where there is a television. The video will pass from the first television (bottom left in the pictures) to the second one, as soon as the location system detects that he is in the new space (the hall).

In both spaces there is a television that can be remotely controlled using the DLNA protocol. Moreover, the service uses a Bluetooth location system available in the laboratory. It is able to calculate the area where the user's smartphone is.

The service is composed of two particular scenes that works together to provide the behaviour. The first one only determines the presence and the state of the objects, then it calls the second one that is responsible for the video playback. The first scene is immutable, it is marked with the unmodifiable label, so when the user moves to another room, it will not be modified. On the other hand, the second ECA rule only has one event and one action and it is always adapted for the new space. The main scene is:

*When I move to another room,
if the DLNA is playing,
and the new space has a `Video.start(videoContent)` capability,
then save the status of the video,
stop the video,
adapt scene 2,
and activate the adapted version of scene 2.*

While the second one is:

*When you are activated,
then play the saved video on the television.*

In this service, we want to prove that the model is suitable to describe the interactions between two scenes and that it can work in real time, using the adaptation like any other capability. The complexity of the scenes will also be high, they will include almost all the features that model provides.

6.4.1 Smart Objects

The objects used in the service are the televisions, the smartphone and the location system. The model of the television is the same of the one seen in the previous service. The other parts are new and they are software components, each of them with its module. The location system is an algorithm that is directly embedded in a module and that receives a Bluetooth signal from a group of beacons and estimates its own position in the space. Another module includes a DLNA controller to manage the televisions and the video content.

For this service we also needed to develop two special modules: one able to communicate with the orchestrator's location manager, to request information about the available objects in the space, the other able to communicate with the adaptation block to force it to adapt a scene, to know when it finishes an adaptation process and what is the resulting scene. These two modules are modelled as “virtual” components. They should be included in the model of the smartphone as for the **ActivityManager** of the previous service, but in this case they have been added to the orchestrator as modules for this service.

6.4.2 Service Discussion

We suppose that the user has already started to watch the video, selecting it through the DLNA controller. This behaviour may be captured and used to enable the first scene, but to simplify the service, we assume that it is already active. The main event that triggers the first scene is a change of location notified by the location system. When it happens, the rule checks if some device is playing a video using the DLNA controller (i.e. if the smartphone is streaming a video to another device). This condition avoids problems and possible errors if no video is playing. The second condition uses the special module we developed for the service. The event implies that the user is now in a new space. Therefore, if the automatic discovery has not been disabled, the location manager looks for new smart objects. The condition asks to the location manager whether it has found a new device able to play a video stream. Since this object (if available) can work as the television, the adaptation process will put it in the second scene. If all the conditions are accomplished, the

first action is executed. It saves the current status of the video, asking it to the DLNA controller. This information is necessary to continue the playback from the same point. Then the video is stopped on the old television, always using the DLNA controller. The first scene is unmodifiable, so it cannot contain a space-dependent object because it would remain always the same. Sending a stop order to the device that is playing a video, using the DLNA controller, avoids this issue. The explicit adaptation of scene 2 is forced. The module that provides this capability is designed to directly modify the ECA rule, without creating copies. In this manner no new behaviours are created every time that the user moves. The last action needs to know the name of the adapted version of scene 2, to be able to activate it. It uses a variable to get that name and it invokes the adaptation process. It automatically generates a callback: the value of the variable blocks the execution of the scene until the adaptation fills that field. The new name is the same of the old one (scene 2), but the use of a variable ensures that the second scene is not activated before the change of its object. The second ECA rule is then activated with the new television. A shared variable permits to pass the state of the video between the two ECA rules, continuing the video playback. The new device starts to play the video (through the DLNA controller), from the point that had been saved in the variable. The code of the scenes is reported in the Listing 6.6 and in the Listing 6.7

Listing 6.6: Pseudo-code of Follow-me scene 1

```

1 Smartphone smartphone = new Smartphone(this);
2 DLNAController dlna = new DLNAController("ExpLabDLNA");
3 LocationSystem location = new LocationSystem("ExpLabLoc");
4
5 ON
6   location.sensor[ProximityLocation].setEventCallback(newSpace != oldSpace);
7 IF
8   dlna.sensor[Video].isPlaying()
9 AND
10  smartphone.LocationManager.exists(newSpace, Video::start())
11 THEN
12   $videoState = dlna.sensor[Video].getValue();
13   dlna.actuator[Video].stop();
14   smartphone.Adaptation.adapt(scene2);
15   smartphone.activate($scene = smartphone.Adaptation.getAdaptedScene(scene2));

```

Listing 6.7: Pseudo-code of Follow-me scene 2

```

1 Smartphone smartphone = new Smartphone(this);
2 Television tv = new Television("LivingRoomTV");
3
4 ON
5   smartphone.setEventCallback(me == active);
6 THEN
7   tv.start($videoState);

```

This service shows an application of the “real-time” adaptation. The scene 2

is modified while the service is running. The adaptation is faster than the DLNA protocol. It takes between two and three seconds to save the information of the video, to stop it and to resume the playback on the second television.

In this service we show that the model is not only suitable to describe the interactions between objects and capabilities, but also between scenes. Moreover, applying the encapsulation, the adaptation process can be seen (and used) as a capability. The scenes of this service are quite complex and they includes almost all the features that the model is able to describe (interactions between objects, capabilities and scenes, variables, complex capabilities and adaptation). Only the automatic recommendation is not used. The next service will be focused on this part.

6.5 Air Traffic Control Station

The last service proposed for the validation is an air traffic control (ATC) station. In this case, we want to show that the model is valid for the automatic recommendation, so we will not analyse scenes and objects in detail as in the previous services. We will focus on the use of the recommender to solve the problem of the portability instead of using the canonical adaptation process based on generalization and specialization.

We created a prototype of a station, to control drones, using a video wall. The interaction with the video wall is performed with gestures and voice commands. The prototype is shown in the Figure 6.13. In this service, we use the smartphone to configure the association between gestures/voice commands and the interactions with the video wall. The system is composed of a kinect+pc block and another video wall+server system. The communication between the two systems is done over the local network. The first system controls the second one sending the commands that permits to the user to interact with the video wall.

Each association command-action is described with a different scene. The list of commands we configured is the following:

- *Fullscreen*: the user points at one of the nine screens and he says the command: “fullscreen” to extend the content of that screen to the entire video wall.
- *Restore*: when the fullscreen is active, the user says: “restore” to come back to the previous configuration.
- *Change configuration*: we implemented four different configurations to assign the contents to the screens. Each configuration has a name. The user says the order: “change to configuration” followed by the name of the configuration to change it.
- *VoIP call*: there is a specific configuration for each drone. This configuration include all data gathered from a particular drone (camera, sensor, mission,

Figure 6.13: Prototype of the ATC station.

etc). It also include a VoIP software to call the pilot. When one of these configurations active, it is always possible to call the pilot. The user says: “call the pilot” to call him. If the pilot answers, the voice recognition is disabled to avoid undesired behaviours.

- *Ends VoIP call*: the screen with the VoIP application shows a big button to drop the call once it answered. Pointing at this button terminates the call and restore the voice command recognition.

The general structure of all the scenes is always an event that catches a user’s command and an action that produces some changes on the graphical interface of the video wall. The scenes are market to belong to the same group.

6.5.1 Service Discussion

The video wall has its own module. It was described in the Chapter 4. It is connected to a server that receives all data from the drones. The configuration of each screen

with the content provided by the drones is also possible through the orchestrator, but to simplify the example we suppose that it is preconfigured on the server.

A Kinect device is installed on top of the video wall and it permits to recognize gestures and voice. The system Kinect+pc provides another module that permits to use the pointing gesture and the voice commands.

With the implementation of the scenes described above, the user can control the video wall. Now let us see what happens when the user goes to another space.

Let us suppose that he moves to another space where there is a screen, that can receive a remote content, and a tablet. The tablet has an application able to detect gestures on its surface (tap, double tap, swipe, pinch, zoom, etc.) and to send them to a remote device through a REST interface. Moreover, the application on the tablet can receive and show a remote content. The input source is selected from within the application.

When the user arrives, his smartphone tries to adapt the scenes to the new space. All of them needs objects able to recognize gestures and voice commands and a video wall to act on it. None of these is available. The only possibility is that both pair of objects (Kinect, tablet and video wall, screen) have some common parent class in the class diagram. Both sensors are “gesture recognition” sensors and both actuators derives from **Video**. The problem for the adaptation is the reduced number of capabilities involved in the scenes (one event and one action). If one of them does not have a similar counterpart in the new space, the adaptation fails. All the ECA rules created for the video wall, except the last one, need a voice command. Thus, the adaptation will fail on all the first four scenes. The last one can be adapted, but it needs a VoIP application making a call to be able to work. If the adaptation fails, the CBR can try to looks for similar scenes in the repository.

Other two methods can be used. The first one is the manual modification of all the scenes. Knowing what they do, the user can manually adapt the behaviour to the smart objects of the new space. Replacing the gestures and the voice commands with some movements on the tablet, and the actions on the video wall with actions on the screens, he can do the adaptation.

For example, users that work in the second space, using tablet and screens, may have shared their scenes to control the drone missions with these devices. Let us suppose that the following scenes have been created and then shared:

- *Fullscreen*: the user sends the content, that he is watching on the tablet, to the screen using a swipe up gesture (another stream is sent to the other device; on the tablet, the content does not change).
- *Restore*: the user restores the content that is on the screen with a swipe down gesture (the content on the tablet is replaced).
- *Change configuration*: the user can change between the contents with a swipe

left or right.

VoIP calls are not included. The user can close the application and make a call with another software manually.

These scenes may be proposed to the user by the *adaptation of previous scenes* block. If they belong to a group, the whole group may be proposed to the user. Using the description included in the scenes, the user can easily understand their behaviour and decide what to do.

Coming back to the CBR, these three ECA rules are stored in a shared repository and they may be used to try to adapt the user's scenes. In this case, the similarity function detects the components that derive from the same parent class. The lack of a voice recognition component only reduces the similarity value, but if that number is not too low for the acceptance limit, the ECA rules stored in the repository may be proposed to the user.

In this example, the last capability helps to differentiate the behaviour in the fullscreen and restore rules (in the first one the action is play, while in the other it is stop). The last scene is not distinguishable by the capabilities. In this case, the parameters differentiate them. Both capabilities and parameters are fundamental to distinguish the scenes and to determine the similarity with the user's ones.

The CBR estimates a similarity ranging from 0.3 and 0.5 for each pair with the same name (fullscreen, restore and change configuration). The values are low, but configuring an acceptance limit of less than 0.3, the three scenes are proposed to the user. He can then accept, modify or discard them.

If also the shared ECA rules belong to the same group, the acceptance value can be increased. If only two scenes have a sufficient similarity value to pass the test, all the group of shared ECA behaviours is proposed to the user.

In this service, we proof the validity of the model for the recommendation of new scenes. It is used to calculate similarity values and the distance between two scenes. This feature is important because it provides a solution to the portability when the adaptation fails.

With this service, all the properties and applications of the model have been tested in a real environment, proving its adequacy to describe smart objects and the relations and the interactions among them.

6.6 Summary

In this chapter we discussed some example of services to show how the system works. The first example was a very simple scene, manually created, to explain the steps that the user must perform to configure an ECA behaviour. In this part, we also introduced the MECCANO mobile application that is used in all the examples. All

the steps have been analysed, from the discovery of the objects until the execution of the scene. The portability of the ECA rule is also tackled. The adaptation of the scene to a new space, passing through generalization and specification process, has been explained. The result is a scene similar to the original one, but not exactly the same. It is a possible output of the system and the user can decide to accept, modify or discard the proposed solution.

In the second example, we use the services of PERSEO to analyse the behaviour of the system with multiple ECA rules instead of only a single scene. We focused our attention on the video behaviour. The adaptation has been discussed explaining the case of a parameter substitution (NFC tag). Parameters are not included in the model, so they need a manual intervention or a recommendation using shared scenes for the substitution.

Then, we presented a simple use of a chain of ECA rules to know which scene was enabled before. Finally, we introduced the resource server that centralizes the assignation of the smart objects to the scenes and it creates priorities on the use of them.

Next, we focused on the Follow-Me service. It permits to a content to follow the user while it moves through different smart spaces. In this example, we pointed out the ability of the system to adapt a scene in (almost) real-time. The ability is used to rapidly detect if there is a screen able to play a video every time that the user goes to a new space. The new smart device is used to adapt the “playing scene” to the new area. In this section, we also provide an example of a scene that uses orchestrator’s internal modules. The first scene contains capabilities that check the state of the orchestrator’s blocks and that control the second ECA rule.

Finally, the last service provides an example of a very complex case. The user wants to adapt the group of ECA rules that he uses to interact with a video wall to a new space where the available objects are very different. The “systematic” adaptation composed of generalization and specification is not able to produce an acceptable solution, in this case. Hence, the other personalization blocks come into play. The easiest solution (for the system) is the manual adaptation of the scenes or the downloading of preconfigured ones. In both cases, the ECA rules (manually adapted by the user or by another person) ends shared in a repository. If the repository has some scenes, the statistical analysis and the CBR may be used. The analysis of shared scenes simply proposes the group of shared ECA behaviours to the user if they have been uploaded/downloaded/used several time in the new space. The CBR calculates the similarity value between the user’s scenes and the repository’s ones. The adaptation is difficult because the ECA rules are not very similar, but configuring a low acceptance limit, the CBR can do the work. The fact that the scenes are grouped helps the CBR to propose the entire groups to the user, even when the similarity between two ECA rules is zero.

All the examples use the system, whose analysis of the scenes is based on the

smart object model. The model is always present in the workflow and it gives to the system a base to create, to adapt, to propose and to execute the ECA rules.

Chapter 7

Conclusions and Future Research Directions

The effort spent on smart objects by the research community and by the technology companies in the past few years, demonstrates how this technology is envisioned to play a key role in the future. The introduction of the Internet of Things enables new services and applications that will make easier our everyday life.

However, despite the increasing interest on smart objects and their growing popularity, many important aspects are still yet to be tackled. One of them is the fragmentation of the devices: the availability of several heterogeneous objects that do not have any mechanism to cooperate among them to improve the user's life. Without this mechanism, smart objects will remain isolated entities unable to provide a real smart space for the users. That mechanism must provide a common manner to discover, configure, control and interact with the smart devices.

In this thesis, we studied and proposed a model to describe smart objects. Using the model, we have implemented that mechanism, defining how the various entities should work to create a personalizable and usable smart space.

This last chapter briefly summarizes the work presented in the thesis, and puts forth a few research and implementation issues for future exploration.

7.1 Contribution Summary

In the chapter 3, we analysed the workflow: the necessary steps that must be followed to integrate a new device in a space and to use it. We identified three big steps: discovery, personalization and interaction. Along the workflow, the system become aware of a new object and its capabilities, it is integrated with the other object by the creation or adaptation of scenes, and finally it is configured and used to produce a part of the space behaviour. Each step of the workflow may be performed in

different ways. Basically they always include a manual operation and one or more automatic modes. We also proposed an architecture with the necessary blocks to cover every possible options for each step of the workflow.

The contribution of this part is the detailed analysis of the necessary steps that permit to integrate an object in a smart space and to use it.

Next, in the chapter 4, we focused on the smart objects as separate entities, looking for a manner to describe them, but having in mind the workflow and the interactions that they should perform. We began observing how a smart device works: what it can do, how it communicates and its internal structure.

From the initial draft, a formal version has been designed, thinking about the problems that these devices currently have, especially the low level of interaction and the heterogeneous interfaces. The second version was an object-oriented model. The smart objects are represented as aggregations of their components: sensors and actuators. Components are arranged in subsets of the same type, represented by abstract classes. This organization provides a common interface for the objects that the developers should follow to increase the level of interaction among the devices. In this part, the main contribution is the OO model that permits to describe a smart objects in term of its components and capabilities.

With the model in mind, in the chapter 5, we analysed each step of the workflow. The features of the OOP have been used on the model and then reinterpreted to apply them on the physical world. An architecture has been proposed to implement each step and the operations performed on objects and scenes.

In this part of the thesis, we first defined the internal structure of a module, created following the proposed model, and of a scene, mashing up the capabilities from various objects. Modules and scenes are the elements that allows a mapping between the model and the real world: a module is a smart device and a scene is a behaviour of the space/objects. Next, we described the blocks of the architecture that implement the workflow. Each step is performed by one or more blocks. Special attention was given to the personalization phase: on how to help the user to adapt his scenes to a new space without manual intervention. For this purpose, we defined the adaptation process, where OOP features, such as inheritance and polymorphism, are applied on the model that describes the objects of a space, and then reflected on the real smart devices. When the adaptation process fails, a case-based reasoner may use the model to look for similar scenes in a repository. If there are not any similar scenes, or the user does not have any scene on his smartphone, a recommender may classify the ECA rules of a repository and propose the most popular ones to the user.

The functional features of the OOP (method, message, etc.) are finally used during the execution. A scene is followed as a computer program and performed, orchestrating the objects and the components involved in it.

The contribution of this part is the use of the model and of the OOP features to act

on the real world, by modifying ECA rules and by executing them.

Finally, we validated the model, the workflow and the architecture through some services. In each service, we saw the behaviour of the architecture's blocks during the workflow and the operations that they perform on the model to personalize a smart space for the user.

7.2 Future Work

There are a number of issues that are raised by this research, but could not be addressed fully. This section investigates some of these issues and points to potential fruitful areas of future research based on this work.

7.2.1 Model

The model is oriented to capabilities and components, but also parameters and variables are used in scenes. Currently their types depends on the type of the component that uses them: a thermal sensor works with temperatures, an accelerometer with accelerations, etc. Modelling also the types of data it is possible to describe also the complex capabilities provided by the smart objects, increasing the integration.

7.2.2 Standardization

Another issue is the need of unification of the class names. As reported in the text, nowadays there are hundreds of type of sensors and actuators. The model is open to the inclusion of any kind on components, but to ensure that the model work perfectly, especially during the personalization, the names of the components should be the same for each object. A thermal sensor should be always called with that term, not temperature sensor or something like that. To solve this problem the model may be extended including any kind of possible components.

7.2.3 Case-Based Reasoning

In the CBR we did not implement the modification in the reuse and the revision step. The revision may be performed by a rule engine to check the scene. The reuse must be completed, doing a fusion of capabilities and parameters in a scene instead of only propose a shared ECA rule as it is. The model can be used for that purpose, for example using an adaptation process similar to the one used in the other part of the system. The use of this approach has been proposed by [Müller and Bergmann \(2015\)](#), applying the generalization-specification to a CBR.

Also the similarity function may be improved. The current function is not able to compare groups of capabilities from a scene with only one capability of the other. Considering for example the case analysed in PERSEO, a video stream may be divided into two streams, one for video and one for audio. The current version of the CBR cannot match the object that plays the stream with the combination of the two devices, although their combined behaviour is the same. The similarity function now returns a 0.5, while considering the combination, it should return something close to 1.

7.2.4 Security

Every time that a scene is shared, all the sensitive data is removed by the modules. This cannot be done during the execution of an ECA rule because an object may need sensitive data. Therefore, a secure communication channel should be established between the orchestrator and the objects. The perfect solution is to always use a secure connection, but some smart devices may not have enough computational power to provide this feature. Another solution may be to use the rule engine during the scene compatibility analysis to recognize which capabilities transport sensitive data and, if the target object is not able to establish a secure connection, to not enable the execution of that scene, or visualizing a warning message asking to the user what to do.

Also the modules and the scenes should use some trust mechanism to ensure that they have not been modified and they cannot produce any potential dangerous behaviour. For the modules, it is quite easy to implement. A connection to a trusted server, where the modules are continuously monitored, may solve the problem. For the scenes it is difficult. They may contain some dangerous behaviours very hard to detect. A rule engine on the repository should check them separately and the possible interaction that many of them used together may produce.

7.2.5 Shared Objects

In a real smart space, it is often possible that more than one user are in the same environment and at least two of them want to use the same object. It is possible to use a resource server to coordinate the access to the devices. The solution proposed in [Bergesio et al. \(2015\)](#) uses a central server. It is limited to a few spaces and to the users that want to use it. It should be implemented with a cloud service or a peer-to-peer architecture where the orchestrators negotiate the access among them. Moreover, the request of a resource should be always mandatory.

7.2.6 Scalability

The download of the modules is currently a bottleneck for the scalability. The modules include all the information about a smart device. To have access to those information, the module must always be downloaded, even when it is already available on the user's smartphone. Not all the information should be packed in the module. If the object is able to provide the data necessary to recognize if a module is already present, then the orchestrator can avoid to re-download it. To solve this problem, the devices must be able to provide pieces of information, implementing some mechanism, more complex than to publish a URL.

7.2.7 Interaction Issues

All the interactions are controlled by the orchestrator. It requires a stable network connection between the smartphone and each smart object. It may be difficult to have this condition, especially in vast spaces or if the user moves quickly. An improvement may be the delegation of the control of the interaction process to one or more objects. Creating clusters of interactions, the smartphone would reduce the number of entities to control. This approach should be an hybrid between the complete mobile-mediated interaction and the "hub-mediated" interaction (Amazon Echo, Google Home, Apple HomePod, etc.).

7.2.8 Coding

The current version of the orchestrator has been only implemented for Android operating system. The development on Apple iPhone was discarded due to the high number of restrictions imposed by the iOS. The most recent versions of iOS are a bit more open than the first ones. Considering the iPhone's market share, it could be interesting to develop the code also for this platform.

On the other hand, Android has been gradually reducing the permissions to the applications throughout the versions. Every year, a part of the code must be adapted to the new version of the operating system, not only the code of the orchestrator, but also all the code of all the modules.

Appendix A

Architecture of the Orchestrator

The internal architecture of the orchestrator is an enhanced version of MECCANO ([Bernardos et al., 2013](#)). It contains all the software components described in the Chapter 5. The orchestrator is currently available for only Android operating system. The initial version of MECCANO has been updated to be compatible with the last version of Android (version 7). During the process, the existing components have been improved with new features and other new components have been added.

MECCANO architecture is device-centric, as it is based on a smartphone application that controls the full workflow: object discovery, scene configuration/personalization and execution.

In this appendix is reported in detail the internal structure of the orchestrator used for the validation. In this case the recommender and the CBR have been implemented in an external server. The smartphone maintains a communications with that components to their services. The UML class diagram is shown in the Figure [A.1](#). According with the colours, it can be divided into three main parts (blue, red and green/orange). The classes drawn in blue are the core of the applications. They contain all the code to control the workflow. Dark blue classes represents the graphical user interface. The red classes are interfaces that permits the integration between the code and the modules that permits to interact with the smart objects. The green and the orange class are two modules. The green one belongs to a smart objects, while the orange one permits to the orchestrator to control some (physical or virtual) component of the smartphone. In this example it is the module that permits to activate and deactivate scenes.

The role of the main classes is now specified. The core of the application is composed of the following main components:

- The **Manager**, which handles modules and scenes lifecycle: it dynamically retrieves scenes from the other components and from the network, saves them in the secondary memory of the smartphone and loads them into memory. It

Figure A.1: Architecture overview

exchanges data among all the components within the smartphone and with the service recommender and the CBR hosted by the infrastructure in order to coordinate the download of new scenes and modules.

- The **Engine**, which drives the scene execution phase. The **Manager** transfers active scenes from the storage component to the engine; it afterwards waits for the events that trigger the scenes and executes them. It also includes the reasoner that checks incompatibilities, inconsistencies and possible dangerous configurations. This component does the checking in two different situations: 1) when configuring a scene, while the user selects each event, condition or action, by blocking the choice of some of them and 2) when the configuration is finished, by examining the whole scene. To prevent errors on operation, the reasoner also checks scenes when they are activated.
- The network interface is the physical component responsible for the communication with objects. It is controlled through the **NetworkHandler** class. During the configuration of a scene, it sends orders to the objects to configure them. During the execution phase it receives events and sends and receives data related to condition checking; it also sends commands to execute actions.

Internally, it communicates directly with the **Manager**.

- The **DiscoveryHandler** is the component that discovers smart objects using some communication technology (e.g. NFC, WiFi, Bluetooth, ZigBee). Objects may directly publish their capability modules or a URL from where they can be downloaded. This class reads and transfers the identifiers to the **Manager**, which downloads the modules.
- The Graphical User Interface (GUI) is composed of several parts. A general structure allows the user to configure, share and download scenes and to download, install and delete modules. Additionally, each module contains some elements to integrate the object capabilities, in particular a set of icons and also the interfaces that are needed to configure specific parameters for the capabilities.
- The **MeccanoCore** is the class that performs the adaptation of the scenes and that exchanges data with the external recommender and the CBR through the **NetworkHandler**. It is the only class that works with the structure of the model obtained from the modules to modify and adapt the scenes. The personalization step of the workflow is almost entirely controlled by this class (GUI excluded).

The communication interfaces (red classes) defines the data structure for the communication between the core of the application and the modules. Together with the communication interface, there are also some classes (e.g. **MeccanoConfigData** class) to organize the internal structure of a scene. These classes simplify coding. **iSensorLister**, **iMeccanoActionListener** and **iModuleOnLine** publish the methods to be implemented by the modules and used by the core to build and execute scenes.

Each external module (green class) is implemented in an APK (Android Application Package) compressed file that contains Java code to implement the common interface and other resources for visualization, i.e. icons and the object GUI. The module publishes the object's capabilities in the module constructor. Modules are loaded at runtime using the dynamic class loader provided by Java and also available for Android. A module can share variables with other modules (e.g. the contact for the last incoming call or the smartphone Bluetooth MAC address) sending the variable to a hash table stored in the **Manager** class. The Appendix B is entirely dedicated to the development and the analysis of a module.

A.1 Scene Description

In the thesis, the scenes are always described using pseudo-code, but in the real implementation they are coded in XML files. The structure of the XML file is divided into two parts: the first part includes the behaviour itself, as a combination of events, conditions and actions; the second part includes a list of the modules necessary for execution. An example of XML is shown in the Listing A.2, it is the first scene used for the validation in the Chapter 6:

*When the main door is opened,
then turn on the light
and set the colour to green.*

The pseudo-code is reported in the Listing A.1.

Listing A.1: Pseudo-code of the scene

```

1 SmartDoor mainDoor = new SmartDoor("mainDoor");
2 WorkLamp smartWLamp = new WorkLamp("light01");
3
4 ON
5   mainDoor.sensor[MagneticSensor].setEventCallback(isOpen == true);
6 THEN
7   smartWLamp.actuator[LEDLight].start();
8   smartWLamp.actuator[LEDLight].control(colour = "green");

```

The XML contains the same information used in the pseudo-code with some additional fields (e.g. hash) used by the system to identify objects, modules and capabilities.

Listing A.2: Example of the XML syntax.

```

1 <?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
2 <MeccanoDef nameEng="DoorLight" author="Luca" descEng="When the..." status="unactive" readonly="no">
3   <Event obj_id="mainDoor" type="event_door">
4     <Param type="param_door">param_door_open</Param>
5   </Event>
6   <Conditions/>
7   <Actions>
8     <Action obj_id="light01" type="action_light_change_state">
9       <Param type="param_on_off">on</Param>
10    </Action>
11    <Action obj_id="light01" type="action_light_change_colour">
12      <Param type="param_colour">green</Param>
13    </Action>
14  </Actions>
15  <Modules>
16    <Module>
17      <SmartObject id="mainDoor">SmartDoor</SmartObject>
18      <Hash>aceb745f149f9e1619360d026dc47bb7</Hash>
19      <Capabilities>
20        <Capability id="event_door">Sensor::DoorStateDetector::MagneticSensor</Capability>

```



```
21     </Capabilities>
22 </Module>
23 <Module>
24   <SmartObject id="light01">WorkLamp</SmartObject>
25   <Hash>dbbb0a16588ca4ff3e63864bdbca8039</Hash>
26   <Capabilities>
27     <Capability id="action_light_change_state">Actuator::Lightning::LEDLight</Capability>
28     <Capability id="action_light_change_colour">Actuator::Lightning::LEDLight</Capability>
29   </Capabilities>
30 </Module>
31 </Modules>
32 </MeccanoDef>
```


Appendix B

Creation of a Smart Object

This annex describes the steps to “smartify” an object. It takes into consideration the smart door of our laboratory used in the Chapter 6. It covers the creation of the hardware and software necessary to provide remote interaction with the smartphone application.

B.1 Model

The first step to create the smart door is the definition of the capabilities that we want the object to have. We will create a door with a sensor able to detect its state and to generate an event when opened or closed. The model provide us some basic capabilities that we should implement, apart from the new ones that we want to add. From the model, we can see the public methods of the `DoorStateDetector`, the class of sensor that we need for our object. They are `startSensing()`,

Figure B.1: Model of the smart door.

`stopSensing()` and `setEventCallback()`.

It is crucial to remember that the important “parts” of a capability is the result that receives the orchestrator (and the perception of the user). The implementation

is irrelevant, any hardware or software component between the component and the orchestrator can perform part of the task to complete a capability.

With this concept in mind, the easiest way to implement `startSensing()` and `stopSensing()` is to do that directly in the module that execute the smartphone. In general, it is the best solution because it does not require any hardware implementation. On the other hand it is a fake implementation: the orchestrator can switch a sensor off, it sees the sensor off, but in fact the sensor is still on. It is better to implement these two methods in hardware when the objects works over batteries or when leaving a sensor on or off can lead to potentially dangerous situations (e.g. overheating, cumulative error of sensors, etc.).

The `setEventCallback()` is the capability that represents the generation of the event, so it is already included in the model.

In order to simplify the description and the code, only those three capabilities will be considered in this appendix. The final version of the model is shown in the Figure [B.1](#).

B.2 Hardware

Having the model in mind, it is possible to continue with the implementation. The next step is the creation of the hardware. It is a very simple circuit composed of an Arduino Fio, equipped with a Wi-Fi module, and a reed switch ([Ellwood, 1941](#)). The reed switch is a very common electrical switch operated by an applied magnetic field. Reed switches are widely used in magnetic sensors.

The connection between the Arduino and the sensor is shown in the Figure [B.2a](#). The Atmega microcontroller on the Arduino has internal pull-up resistors (resistors that connect to power internally), we use that resistor, so it is not necessary to add a new one to the circuit.

The Arduino is connected to a USB power supply, so we do not have to worry about the power consumption during the development of the software.

B.3 Arduino Code

The Arduino board receives the signals from the sensor, notifies the state changes and saves the current state. The signal that comes from the sensor must be filtered to avoid problems with the switch bounce. The network communication should be implemented with a notification protocol, we initially thought to use the MQTT. Unfortunately, the Arduino Fio has very little memory and the library needed to control the Wi-Fi module occupies a big part of it. Hence, we could not integrate the MQTT directly in the Arduino and we had to use an external server to do that.

(a) Sketch of the circuit.

(b) Sensor mounted on the door.

Figure B.2: Hardware used to create the smart door.

The code is reported in the Listing B.1.

Listing B.1: Arduino code

```

1  #include <WiFlyHQ.h>
2
3  #define NUM_READING 8
4  #define LIMIT 6
5  #define HYSTERESIS 2
6
7  const int pinSensor = 3;
8
9  boolean isOpen = false;
10
11 volatile int sensorReadings[NUM_READING];
12 volatile static int counter = 0;
13 int value = 0;
14
15 const char mySSID[] = "*****";
16 const char myPassword[] = "*****";
17 const uint16_t myPort = 50000;
18
19 void sendData(int val);
20 void joinWifly();
21
22 WiFly wifly;
23
24 void setup() {
25     pinMode(pinSensor, INPUT_PULLUP);
26
27     Serial.begin(9600);
28     wifly.begin(&Serial, NULL);
29
30     joinWifly();
31
32     if (wifly.isConnected()) {
33         wifly.close();
34     }
35

```

```
36 value = 0;
37 for (int i = 0; i < NUM_READING; i++) {
38     sensorReadings[i] = digitalRead(pinSensor);
39     value += sensorReadings[i];
40     delay(100);
41 }
42
43 if (value > LIMIT) {
44     isOpen = false;
45     sendData(1);
46 } else {
47     isOpen = true;
48     sendData(0);
49 }
50 }
51
52 void loop() {
53     counter = (counter + 1) % NUM_READING;
54     sensorReadings[counter] = digitalRead(pinSensor);
55
56     value = 0;
57     for (int i = 0; i < NUM_READING; i++) {
58         value += sensorReadings[i];
59     }
60
61     if (value > LIMIT && isOpen) {
62         //Door is open
63         isOpen = false;
64         sendData(1);
65     } else if (value >= 0 && value <= LIMIT - HYSTERESIS && !isOpen) {
66         //Door is closed
67         isOpen = true;
68         sendData(0);
69     }
70
71     delay(100);
72 }
73
74 void sendData(int val) {
75     joinWifly();
76
77     if (wifly.open("192.168.150.20", myPort)) {
78         wifly.println(val);
79         wifly.close();
80     }
81 }
82
83 void joinWifly() {
84     /* Join wifi network if not already associated */
85     if (!wifly.isAssociated()) {
86         /* Setup the WiFly to connect to a wifi network */
87         wifly.setSSID(mySSID);
88         wifly.setPassphrase(myPassword);
89         wifly.enableDHCP();
90         wifly.setDNS("8.8.8.8");
91         wifly.setProtocol(WIFLY_PROTOCOL_TCP);
92         wifly.setDeviceID("Wifly-Door");
93         wifly.join();
94     }
95 }
```

To avoid wrong reading from the sensor, this value must be stable during at least 6 different samples. An initial reading is performed during the setup to correctly configure the initial state of the door. Every time that the state changes, the new value is stored in `isOpen` and a packet containing 1 (opened) or 0 (closed) is sent to a server. The rest of the variables and the code is used to control the access to the Wi-Fi module mounted on the Arduino.

B.4 Notification Server

We want to use the MQTT protocol to notify the events coming from the door. As we seen above, the Arduino Fio has not enough memory to contain the driver for the Wi-Fi module, our code for the smart door and the code for the MQTT. The Arduino is only able to open a TCP/IP connection with the server located at 192.168.150.20 and then to send the value that represents the state of the door. The role of that server is to convert that packet into a MQTT notification for the orchestrator and for other entities.

On the same machine where the notification server will run there is a standard MQTT-Broker already installed and running. The notification server will publish its events to this MQTT-Broker. All the devices subscribed to that broker and to a specific topic (`door`) will receive the notifications.

The code of the notification server has been implemented in JAVA. Its behaviour is very simple. It waits for a packet coming from the Arduino. It parses the one-byte message that contains the state of the door. Finally, it publishes a new message to the MQTT-Broker, forwarding the state of the door. The code is reported in the Listing B.2.

Listing B.2: Java code of the notification server

```
1 public class DoorMagneticSensor {
2     public static void main(String[] args) throws Exception {
3         int port = 50000;
4         int oldRead = -1;
5         ServerSocket serverSocket = new ServerSocket(port);
6
7         MqttClient client;
8
9         try {
10             while (true) {
11                 Socket socket = serverSocket.accept();
12                 BufferedReader reader = new BufferedReader(new InputStreamReader(socket
13                     .getInputStream()));
14
15                 int newRead = reader.read();
16
17                 if (newRead != oldRead) {
18                     oldRead = newRead;
19
20                     client = new MqttClient("tcp://localhost:1883", "door");
```

```

21     client.connect();
22     MqttMessage message = new MqttMessage();
23
24     if (newRead == 49) {
25         message.setPayload(("open " + socket.getRemoteSocketAddress()
26             .toString()).getBytes());
27         client.publish("door", message);
28         client.disconnect();
29     } else if (newRead == 48) {
30         message.setPayload(("close " + socket.getRemoteSocketAddress()
31             .toString()).getBytes());
32         client.publish("door", message);
33         client.disconnect();
34     }
35 }
36 }
37 } catch (Exception e) {
38     serverSocket.close();
39     e.printStackTrace();
40 }
41 }
42 }

```

B.5 Module

The module has been developed for Android, so the programming language is JAVA. In the Listing B.3 is reported all the code of the module of the smart door. The code is quite long and complex, even if in this example we only consider one capability. It implements all the methods included in the generic `IModule` interface and, in this case, the methods of the `MqttCallback` that permits to receive the events from the MQTT-Broker. Most of the methods implements GUI elements and secondary features, while others are left unimplemented because related to conditions and actions that the smart door does not have. It only has an event. The lines of the code where there are the three methods defined in the model are 39, 68 and 90.

As reported at the beginning of this appendix, the implementation of `startSensing()` (line 35) and `stopSensing()` (line 86) may be done anywhere between the object and the orchestrator. We do it in the module by simply subscribe or unsubscribe the orchestrator to the MQTT-Broker. In this manner it will receive or stop receiving the events from the smart door.

The `setEventCallback()` is represented by the method that begins from line 68. As explained in the thesis, the `setEventCallback()` is not a real method. It is a manner to represent the callback to a function every time that an event occurs. In this case, the event is encapsulated in a MQTT message, so the callback must be included in this `messageArrived()` method. The two possible clause of the `setEventCallback()` are: the door is open or the door is closed. the two ifs at lines 76 and 81 checks the event with the clause configured by the user (the `mcd` variable). Then, both of them call the `onSensorActivated()` method. It is the

global method that every object when a new event happens. To distinguish the event, it is directly passed as a parameter of that method. During the adaptation process, the orchestrator will check for the substitution of the `setEventCallback()` the `onSensorActivated()` methods and possibly their parameters.

The strings at the beginning of the code represent the manifest of the module. The manifest is built in the constructor of the module. The orchestrator obtains these values by calling all the setter methods that return events, conditions, actions, parameters, variables and chains of inheritance (e.g. the `getId()` is at line 120). In this example only an event is represented with two possible parameters and one chain of inheritance that describes the relation between the door and its magnetic sensor.

Other files contains the rest of the GUI, the Android manifest where are defined the permission requested for the execution of the code and some other minor features.

Listing B.3: Android code of the module

```

1 public class Module implements IModule, MqttCallback {
2     private MeccanoActivity mActivity;
3     private static final String ID = "mainDoor";
4     private static final String EVENT_DOOR = "event_door";
5     private static final String PARAMETER_DOOR = "param_door";
6     private static final String PARAMETER_DOOR_OPEN = "param_door_open";
7     private static final String PARAMETER_DOOR_CLOSE = "param_door_close";
8     private static final String INHERITANCE = "Sensor::DoorStateDetector" +
9         ">::MagneticSensor";
10
11     private ISensorListener mListener;
12     private Vector<String> events;
13     private MqttClient client;
14     private static final String address = "tcp://192.168.150.20:1883";
15     private static final String TOPIC = "door";
16
17     private int subscribers = 0;
18
19     private Vector<MeccanoConfigData> paramsEvent = null;
20     private Vector<String> inheritanceChains = null;
21     private MeccanoConfigData mcd;
22
23     public Module() {
24         events = new Vector<String>();
25         events.add(EVENT_DOOR);
26
27         paramsEvent = new Vector<MeccanoConfigData>();
28         inheritanceChains = new Vector<String>();
29     }
30
31     @Override
32     public void configModule(MeccanoActivity activity, Service service,
33         ISensorListener listener) {
34         this.mActivity = activity;
35         this.mListener = listener;
36     }
37
38     @Override
39     public void startSensing(MeccanoConfigData event) {
40         if (event.getId().equals(EVENT_DOOR)) {

```

```

41     paramsEvent.add(event);
42
43     if (subscribers == 0)
44     try {
45         Random ran = new Random(Calendar.getInstance().getTimeInMillis());
46         String uniqueId = ID + ran.nextInt(30000);
47         client = new MqttClient(address, uniqueId, null);
48         client.connect();
49         client.subscribe(TOPIC, 1);
50         client.setCallback(this);
51     } catch (Exception e) {
52         e.printStackTrace();
53     }
54
55     subscribers++;
56 }
57
58
59 @Override
60 public void connectionLost(Throwable cause) {
61     Log.d(ID, "Connnection lost: " + cause.toString());
62 }
63
64 @Override
65 public void deliveryComplete(IMqttDeliveryToken token) { }
66
67 @Override
68 public void messageArrived(String topic, MqttMessage message) throws Exception {
69     if (topic.equals(TOPIC)) {
70         String msg = new String(message.getPayload());
71
72         Iterator<MeccanoConfigData> i = paramsEvent.listIterator();
73         while (i.hasNext()) {
74             MeccanoConfigData mcd = (MeccanoConfigData) i.next();
75
76             if (msg.contains("open") && mcd.getParams().get(PARAMETER_DOOR)
77                 .equals(PARAMETER_DOOR_OPEN)) {
78                 mListener.onSensorActivated(new MeccanoConfigData(mcd));
79             }
80
81             if (msg.contains("close") && mcd.getParams().get(PARAMETER_DOOR)
82                 .equals(PARAMETER_DOOR_CLOSE)) {
83                 mListener.onSensorActivated(new MeccanoConfigData(mcd));
84             }
85         }
86     }
87 }
88
89 @Override
90 public void stopSensing(MeccanoConfigData event) {
91     if (event.getId().equals(EVENT_DOOR)) {
92         paramsEvent.remove(event);
93         subscribers--;
94
95         if (subscribers == 0)
96         try {
97             if (client.isConnected()) {
98                 client.unsubscribe(TOPIC);
99                 client.disconnect();
100             }
101
102             client.close();

```

```

103         } catch (Exception e) {
104             e.printStackTrace();
105         }
106     }
107 }
108
109 @Override
110 public boolean check(MeccanoConfigData param) {
111     return false;
112 }
113
114 @Override
115 public void execute(IMeccanoActionListener listener, MeccanoConfigData action) {
116     // No actions
117 }
118
119 @Override
120 public String getId() {
121     return ID;
122 }
123
124 @Override
125 public Vector<String> getEventCodes() {
126     return events;
127 }
128
129 @Override
130 public Vector<String> getConditionCodes() {
131     //No conditions
132     return null;
133 }
134
135 @Override
136 public Vector<String> getActionCodes() {
137     //No actions
138     return null;
139 }
140
141 @Override
142 public Vector<String> getInheritanceChains() {
143     return inheritanceChains;
144 }
145
146 @Override
147 public void configEvent(String eventType, ViewGroup view,
148     MeccanoConfigData oldConfig) {
149     LinearLayout lv = new LinearLayout(mActivity);
150     lv.setOrientation(LinearLayout.VERTICAL);
151     lv.setBackgroundColor(Color.BLACK);
152     view.addView(lv);
153     TextView tv = new TextView(mActivity);
154     tv.setText("What type of event?");
155     lv.addView(tv);
156     RadioGroup rGroup = new RadioGroup(mActivity);
157     final RadioButton plugRButton = new RadioButton(mActivity);
158     plugRButton.setText("Open the door");
159     final RadioButton unplugRButton = new RadioButton(mActivity);
160     unplugRButton.setText("Close the door");
161     rGroup.addView(plugRButton, new ViewGroup.LayoutParams(
162         ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.WRAP_CONTENT));
163     rGroup.addView(unplugRButton, new ViewGroup.LayoutParams(
164         ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.WRAP_CONTENT));

```

```

165     lv.addView(rGroup);
166
167     if ((oldConfig != null) && (EVENT_DOOR.equals(oldConfig.getId())))
168     if (PARAMETER_DOOR_OPEN.equals(oldConfig.getParams().get(PARAMETER_DOOR)))
169     plugRButton.setChecked(true);
170     else if (PARAMETER_DOOR_CLOSE.equals(oldConfig.getParams().get(PARAMETER_DOOR)))
171     unplugRButton.setChecked(true);
172
173     Button acceptButton = new Button(mActivity);
174     acceptButton.setText("Ok");
175     lv.addView(acceptButton);
176
177     acceptButton.setOnClickListener(new OnClickListener() {
178
179         @Override
180         public void onClick(View v) {
181             mcd = new MeccanoConfigData(EVENT_DOOR);
182
183             if (plugRButton.isChecked())
184             mcd.addParamNameAndValue(PARAMETER_DOOR, PARAMETER_DOOR_OPEN);
185             else if (unplugRButton.isChecked())
186             mcd.addParamNameAndValue(PARAMETER_DOOR, PARAMETER_DOOR_CLOSE);
187             else
188             return;
189
190             mActivity.setMeccanoConfig(mcd);
191         }
192     });
193 }
194
195 @Override
196 public void configCondition(String conditionType, ViewGroup view,
197     MeccanoConfigData oldConfig) {
198     //No conditions
199 }
200
201 @Override
202 public void configAction(String actionType, ViewGroup view,
203     MeccanoConfigData oldConfig) {
204     // No actions
205 }
206
207 @Override
208 public void beReadyToCheck(MeccanoConfigData meccanoConfigData) { }
209
210 @Override
211 public void stopBeingReadyToCheck(MeccanoConfigData meccanoConfigData) { }
212
213 @Override
214 public boolean isConfigured(MeccanoConfigData mEvent) {
215     return
216     mEvent != null &&
217     (
218     mEvent.getId().equals(EVENT_DOOR) &&
219     (
220     PARAMETER_DOOR_OPEN.equals(mEvent.getParams().get(PARAMETER_DOOR))
221     ||
222     PARAMETER_DOOR_CLOSE.equals(mEvent.getParams().get(PARAMETER_DOOR))
223     )
224     );
225 }
226

```

```
227  @Override
228  public void exitWithoutConfig() { }
229
230  @Override
231  public void updateMeccanoActivity(MeccanoActivity activ) {
232      this.mActivity = activ;
233  }
234
235  @Override
236  public Object getVariable(String code) {
237      return null;
238  }
239
240  @Override
241  public void removeSensitiveParams(MeccanoConfigData mcd) { }
242
243  @Override
244  public boolean isEconomicallyExpensive(String code) {
245      return false;
246  }
247
248  @Override
249  public boolean isBatteryExpensive(String code) {
250      return false;
251  }
252 }
```

B.6 Discovery

For the discovery we will use an NFC tag. It is hidden in a sticker and attached to the door, it is visible in the Figure B.3. In this case we use a manual discovery. The module is stored on a remote server and in the NFC tag it is written the URL to reach the module at that server and the identifier of the door (the same value provided by the `getId()` of the code). The URL is the following.

`http://192.168.150.127/meccano/module/DoorModule.apk?id=mainDoor`

The position is not reported in this server, so it will be the location manager of the orchestrator who will assign a position to the object. This choice is dictated by the fact that there is a location system in the lab, so it is not necessary to include the position of the door manually in the server.

Figure B.3: NFC sticker attached to the door.

References

- Alsbou, T. A., Hammoudeh, M., Bandar, Z., and Nisbet, A. An overview and classification of approaches to information extraction in wireless sensor networks. In *Proceedings of the 5th International Conference on Sensor Technologies and Applications (SENSORCOMM'11)*, pages 255–260, August 2011.
- Apple Inc. Bonjour, August 2002. URL <https://www.apple.com/support/bonjour/>. Retrieved: 2016-30-01.
- Apple Inc. AirPlay, June 2011. URL <https://support.apple.com/en-gb/HT204289>. Retrieved: 2016-30-01.
- Apple Inc. Homekit, June 2014. URL <https://developer.apple.com/homekit/>. Retrieved: 2016-09-03.
- Arduino. Arduino Fio, 2010. URL <https://www.arduino.cc/en/Main/ArduinoBoardFio>. Retrieved: 2017-06-24.
- Armstrong, D. J. The Quarks of Object-oriented Development. *Commun. ACM*, 49(2):123–128, February 2006. ISSN 0001-0782. doi: 10.1145/1113034.1113040. URL <http://doi.acm.org/10.1145/1113034.1113040>.
- Ashton, K. That ‘internet of things’ thing. *RFID Journal*, 22(7):97–114, 2009.
- Augusto, J. C. and Nugent, C. D. *Designing Smart Homes: The Role of Artificial Intelligence*, volume 4008. Springer, 2006.
- Bandyopadhyay, D. and Sen, J. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications*, 58(1):49–69, 2011.
- Beigl, M. Point & click-interaction in smart environments. In Gellersen, H.-W., editor, *Handheld and Ubiquitous Computing*, volume 1707 of *Lecture Notes in Computer Science*, pages 311–313. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66550-2. doi: 10.1007/3-540-48157-5_31. URL http://dx.doi.org/10.1007/3-540-48157-5_31.

- Bellavista, P., Cardone, G., Corradi, A., and Foschini, L. Convergence of MANET and WSN in IoT urban scenarios. *Sensors Journal, IEEE*, 13(10):3558–3567, 2013.
- Bergesio, L., Marquínez, Í., Bernardos, A. M., Besada, J. A., and Casar, J. R. PERSEO: a system to Personalize the Environment Response through Smart phonEs and Objects. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 640–645. IEEE, 2013.
- Bergesio, L., Bernardos, A. M., Marquínez, Í., Besada, J. A., and Casar, J. R. PERSEO: A system to Personalize smart EnviRonments through Smartphones, sEnsors and media Objects. *Journal of Ambient Intelligence and Smart Environments*, 7(6):777–804, 2015.
- Bergmann, R. On the Use of Taxonomies for Representing Case Features and Local Similarity Measures. In Gierl, L. and Lenz, M., editors, *Proceedings of the 6th German Workshop on Case-Based Reasoning (GWCBR'98)*, 1998. URL http://www.wi2.uni-trier.de/publications/1998_BergmannGwcbbr.pdf.
- Bergmann, R., Althoff, K.-D., Minor, M., Reichle, M., and Bach, K. Case-Based Reasoning – Introduction and Recent Developments. *KI - Künstliche Intelligenz, German Journal on Artificial Intelligence - Organ des Fachbereiches "Künstliche Intelligenz" der Gesellschaft für Informatik e.V. (KI)*, 23(1):5–11, 1 2009.
- Bernardos, A. M., Casar, J. R., Cano, J., and Bergesio, L. Enhancing interaction with smart objects through mobile devices. In *Proceedings of the 9th ACM international symposium on Mobility management and wireless access*, pages 199–202. ACM, 2011.
- Bernardos, A. M., Bergesio, L., Iglesias, J., and Casar, J. R. MECCANO: a mobile-enabled configuration framework to coordinate and augment networks of smart objects. *J. UCS*, 19(17):2503–2525, 2013.
- Bernardos, A. M., Muñoz, A., Bergesio, L., Besada, J. A., and Casar, J. R. A multimodal interaction system for big displays. In *Proceedings of the 6th ACM International Symposium on Pervasive Displays, PerDis '17*, pages 28:1–28:2, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5045-7. doi: 10.1145/3078810.3084353. URL <http://doi.acm.org/10.1145/3078810.3084353>.
- Bernstein, P. A. Middleware: A model for distributed system services. *Commun. ACM*, 39(2):86–98, February 1996. ISSN 0001-0782. doi: 10.1145/230798.230809. URL <http://doi.acm.org/10.1145/230798.230809>.

- Black, A. P. Object-oriented programming: some history, and challenges for the next fifty years. *Information and Computation*, 231:3–20, 2013.
- Blom, J. Personalization: a taxonomy. In *CHI’00 extended abstracts on Human factors in computing systems*, pages 313–314. ACM, 2000.
- Boren, Z. D. There are officially more mobile devices than people in the world, 2015. URL <http://www.independent.co.uk/life-style/gadgets%2Dand%2Dtech/news/there%2Dare%2Dofficially%2Dmore%2Dmobile%2Ddevices%2Dthan%2Dpeople%2Din%2Dthe%2Dworld%2D9780518.html>. The Independent, Retrieved: 2016-09-07.
- Bormann, D. C., Hartke, K., and Shelby, Z. The Constrained Application Protocol (CoAP). IETF RFC 7252, October 2015. URL <https://rfc-editor.org/rfc/rfc7252.txt>.
- Bowen, J. P. and Filippini-Fantoni, S. Personalization and the web from a museum perspective. In *Museums and the Web*, volume 4, 2004.
- Braynov, S. Personalization and customization technologies. *The Internet Encyclopedia*, 2003.
- Broll, G., Rukzio, E., Paolucci, M., Wagner, M., Schmidt, A., and Hussmann, H. PerCI: Pervasive service interaction with the internet of things. *Internet Computing, IEEE*, 13(6):74–81, Nov 2009. ISSN 1089-7801. doi: 10.1109/MIC.2009.120.
- Castagna, G. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, May 1995. ISSN 0164-0925. doi: 10.1145/203095.203096. URL <http://doi.acm.org/10.1145/203095.203096>.
- Cheshire, S. and Krochmal, M. DNS-Based Service Discovery. IETF RFC 6763, October 2015a. URL <https://rfc-editor.org/rfc/rfc6763.txt>.
- Cheshire, S. and Krochmal, M. Multicast DNS. IETF RFC 6762, November 2015b. URL <https://rfc-editor.org/rfc/rfc6762.txt>.
- CityHome. Places of living and work, 2014. URL <http://cp.media.mit.edu/places%2Dof%2Dliving%2Dand%2Dwork/>. MIT Media Lab. Retrieved: 2016-01-11.
- Cook, D. and Das, S. *Smart Environments: Technology, Protocols and Applications*. Wiley-Interscience, 2005. ISBN 0-471-54448-5.
- Dahl, O.-J. *SIMULA 67 Common Base Language*. Norwegian Computing Center, Publication No. S-2, May 1968. ISBN B0007JZ9J6.

- Dahl, Y. Redefining smartness: The smart home as an interactional problem. In *Intelligent Environments, 2008 IET 4th International Conference on*, pages 1–8, July 2008.
- De Ipiña, D. L., Vázquez, J. I., García, D., Fernández, J., García, I., Sainz, D., and Almeida, A. Emi2lets: A reflective framework for enabling ami. *J. UCS*, 12(3): 297–314, 2006.
- Deering, S. and Hinden, R. Internet Protocol, Version 6 (IPv6) Specification. IETF RFC 2460, December 1998. URL <http://tools.ietf.org/html/rfc2460>.
- DLNA Organization. DLNA Guidelines, March 2014. URL <http://www.dlna.org>.
- Driscoll, D. and Mensch, A. Devices Profile for Web Services Version 1.1. Technical report, OASIS, July 2009. URL <http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf>.
- Drossos, N. and Kameas, A. Building composeable smart objects. In *Proceedings of the 1st International Workshop on Design and Integration Principles for Smart Objects. DIPSO*, 2007.
- Ellwood, W. B. Electromagnetic switch, December 2 1941. URL <https://www.google.com/patents/US2264746>. US Patent 2,264,746.
- eMarketer. 2 Billion Consumers Worldwide to Get Smart(phones) by 2016, 2014. URL <http://www.emarketer.com/Article/2%2DBillion%2DConsumers%2DWorldwide%2DSmartphones%2Dby%2D2016/1011694>. Retrieved: 2016-09-07.
- Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10*, pages 179–194, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-985-5. doi: 10.1145/1814433.1814453. URL <http://doi.acm.org/10.1145/1814433.1814453>.
- Fan, H. and Poole, M. S. What is personalization? Perspectives on the design and implementation of personalization in information systems. *Journal of Organizational Computing and Electronic Commerce*, 16(3-4):179–202, 2006.
- Fernández-Montes, A., Ortega, J., Sánchez-Venzalá, J., and González-Abril, L. Software reference architecture for smart environments: Perception. *Computer Standards & Interfaces*, 36(6):928–940, 2014.
- Follow-Me. MOBILOC and Follow-Me, April 2015. URL <https://www.youtube.com/watch?v=WrbHFu65Bn0>. Retrieved: 2016-10-30.

- Forum UPnP™. UPnP™ Device Architecture 1.1, October 2008. URL <http://www.upnp.org/>.
- Fujinami, K., Kawsar, F., and Nakajima, T. Awaremirror: A personalized display using a mirror. In Gellersen, H.-W., Want, R., and Schmidt, A., editors, *Pervasive Computing*, volume 3468 of *Lecture Notes in Computer Science*, pages 315–332. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26008-0. doi: 10.1007/11428572_19. URL http://dx.doi.org/10.1007/11428572_19.
- Gartner Inc. Gartner Says 8.4 Billion Connected “Things” Will Be in Use in 2017, Up 31 Percent From 2016, February 2017. URL <http://www.gartner.com/newsroom/id/3598917>. Gartner Press Release. Retrieved: 2017-05-23.
- Google. Eddystone, July 2015. URL <https://developers.google.com/beacons/eddystone>. Retrieved: 2017-06-24.
- Google. Android Things, 2016. URL <https://developer.android.com/things/index.html>. Retrieved: 2017-06-14.
- Gorman, L. O. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040, 2003.
- Goumopoulos, C. and Kameas, A. Ambient ecologies in smart homes. *The Computer Journal*, 52(8):922–937, 2009a.
- Goumopoulos, C. and Kameas, A. Smart objects as components of ubicomp applications. *International Journal of Multimedia and Ubiquitous Engineering*, 4(3): 1–20, 2009b.
- Hardy, R. and Rukzio, E. Touch & interact: Touch-based interaction of mobile phones with displays. In *Proceedings of the 10th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI ’08, pages 245–254, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-952-4. doi: 10.1145/1409240.1409267. URL <http://doi.acm.org/10.1145/1409240.1409267>.
- Hernández, M. E. P. and Reiff-Marganiec, S. Classifying smart objects using capabilities. In *Smart Computing (SMARTCOMP), 2014 International Conference on*, pages 309–316, November 2014. doi: 10.1109/SMARTCOMP.2014.7043873.
- Hightower, J. and Borriello, G. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, August 2001. ISSN 0018-9162. doi: 10.1109/2.940014. URL <http://dx.doi.org/10.1109/2.940014>.

- Honkola, J., Laine, H., Brown, R., and Tyrkko, O. Smart-M3 information sharing platform. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 1041–1046, June 2010. doi: 10.1109/ISCC.2010.5546642.
- House_n. House_n projects, 1999. URL http://web.mit.edu/cron/group/house_n/index.html. MIT Department of Architecture Research Consortium. Retrieved: 2016-06-26.
- Howard, M., Bezault, E., Meyer, B., Colnet, D., Stapf, E., Arnout, K., and Keller, M. Type-safe covariance: Competent compilers can catch all catcalls. *Swiss Federal Institute for Technology Zürich*, April 2003. Technical report.
- Hunt, J. *Java and Object Orientation: An Introduction*. Springer London, 2002. ISBN 9781852335694. URL <https://books.google.it/books?id=a4a4PtXj1BsC>.
- IFTTT. Connect the apps you love, 2011. URL <https://ifttt.com/>. Retrieved: 2016-09-07.
- Iglesias, J., Bernardos, A., Bergesio, L., Cano, J., and Casar, J. Towards a lightweight mobile semantic-based approach for enhancing interaction with smart objects. In Omatu, S., De Paz Santana, J. F., González, S. R., Molina, J. M., Bernardos, A. M., and Rodríguez, J. M. C., editors, *Distributed Computing and Artificial Intelligence*, volume 151 of *Advances in Intelligent and Soft Computing*, pages 185–196. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28764-0. doi: 10.1007/978-3-642-28765-7_22. URL http://dx.doi.org/10.1007/978-3-642-28765-7_22.
- Intel. Intel IoT Platform, November 2015. URL <http://www.intel.com/content/www/us/en/internet-of-things/overview.html>. Retrieved: 2016-02-08.
- IoT-Architecture Consortium. Architectural Reference Model for the IoT - (ARM) Introduction Booklet, 2013. URL http://www.iot-a.eu/public/public-documents/copy_of_d1.2/at_download/file. Retrieved: 2016-01-22.
- Kallmann, M. and Thalmann, D. Modeling objects for interaction tasks. In *Proc. Eurographics Workshop on Animation and Simulation*, pages 73–86, 1998.
- Kawamoto, R., Emori, T., Sakata, S., Furuhashi, K., Yuasa, K., and Hara, S. DLNA-ZigBee Gateway Architecture and Energy Efficient Sensor Control for Home Networks. In *Mobile and Wireless Communications Summit, 2007. 16th IST*, pages 1–5, July 2007. doi: 10.1109/ISTMWC.2007.4299193.

- Kawsar, F., Fujinami, K., and Nakajima, T. Prottoy middleware platform for smart object systems. *International Journal of Smart Home*, 2(3):1–18, 2008.
- Kay, A. C. The early history of Smalltalk. In *The second ACM SIGPLAN conference on History of programming languages, HOPL-II*, pages 511–598, New York, NY, USA, 1993. ACM.
- Kim, J. E., Boulos, G., Yackovich, J., Barth, T., Beckel, C., and Mosse, D. Seamless integration of heterogeneous devices and access control in smart homes. In *Intelligent Environments (IE), 2012 8th International Conference on*, pages 206–213, June 2012. doi: 10.1109/IE.2012.57.
- Kinect. Kinect for Windows Sensor Components and Specifications, March 2012. URL <https://msdn.microsoft.com/en-us/library/jj131033.aspx>. Retrieved: 2016-10-30.
- Klauck, R. and Kirsche, M. *Ad-hoc, Mobile, and Wireless Networks: 11th International Conference, ADHOC-NOW 2012, Belgrade, Serbia, July 9-11, 2012. Proceedings*, chapter Bonjour Contiki: A Case Study of a DNS-Based Discovery Service for the Internet of Things, pages 316–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31638-8. doi: 10.1007/978-3-642-31638-8_24. URL http://dx.doi.org/10.1007/978-3-642-31638-8_24.
- Klyne, G. and Carroll, J. J. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004. URL <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
- KNX Association and others. KNX Standard. *System Specifications, KNXnet/IP, Routing*, 2004.
- Kortuem, G., Kawsar, F., Fitton, D., and Sundramoorthy, V. Smart objects as building blocks for the Internet of things. *Internet Computing, IEEE*, 14(1):44–51, Jan 2010. ISSN 1089-7801. doi: 10.1109/MIC.2009.143.
- Kuhn, H. W. The hungarian method for the assignment problem. In *Naval Research Logistic Quarterly*, volume 2, pages 83–97. Kuhn’s original publication, 1955.
- Lampe, M., Hinske, S., and Brockmann, S. Mobile device based interaction patterns in augmented toy environments. In *Pervasive 2006 Workshop Proceedings (Third International Workshop on Pervasive Gaming Applications, PerGames 2006)*, 2006.
- LIFX. LIFX Color 1000 info sheet, August 2016. URL <http://www.lifx.com/pages/color-1000-info-sheet>. Retrieved: 2016-08-30.

- Liskov, B. Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM. ISBN 0-89791-266-7. doi: 10.1145/62138.62141. URL <http://doi.acm.org/10.1145/62138.62141>.
- Liskov, B. H. and Wing, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. ISSN 0164-0925. doi: 10.1145/197320.197383. URL <http://doi.acm.org/10.1145/197320.197383>.
- Liu, L., Kuo, S. M., and Zhou, M. Virtual sensing techniques and their applications. In *Networking, Sensing and Control, 2009. ICNSC'09. International Conference on*, pages 31–36. IEEE, 2009.
- Lunden, I. IFTTT Launches 3 “Do” Apps To Automate Photo Sharing, Tasks, Notes; Rebrands Main App “IF”, 2015. URL <http://techcrunch.com/2015/02/19/ifttt%2Dlaunches%2D3%2Ddo%2Dapps%2Dto%2Dautomate%2Dphoto%2Dsharing%2Dtasks%2Dnotes%2Drebrands%2Dmain%2Dapp%2Dif/>. TechCrunch, Retrieved: 2016-09-07.
- Lutron. Lutron caséta wireless, 2014. URL http://www.lutron.com/TechnicalDocumentLibrary/3672499a_Caseta_Brochure.pdf. Lutron’s devices brochure. Retrieved: 2016-06-26.
- Manovich, L. The poetics of urban media surfaces. *First Monday*, 2006.
- MavHome. Managing an adaptive versatile home, 2001. URL <http://ailab.wsu.edu/mavhome/index.html>. Washington State University and the University of Texas at Arlington. Retrieved: 2016-06-26.
- Michael Teeuw. MagicMirror². The open source modular smart mirror platform., April 2014. URL <https://magicmirror.builders/>. Retrieved: 2017-01-23.
- Miles, R. and Hamilton, K. *Learning UML 2.0*. O’Reilly Media, Inc., 2006. ISBN 0596009828.
- Misra, P., Simmhan, Y., and Warrior, J. Towards a practical architecture for the next generation internet of things. *arXiv preprint arXiv:1502.00797*, 2015.
- Montenegro, G., Schumacher, C., and Kushalnagar, N. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. IETF RFC 4919, October 2015. URL <https://rfc-editor.org/rfc/rfc4919.txt>.

- Müller, G. and Bergmann, R. Generalization of Workflows in Process-Oriented Case-Based Reasoning. In *Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2015*, pages 391–396, Hollywood, Florida, USA, May 2015. AAAI. URL <http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS15/paper/view/10369>.
- Musto, C., Narducci, F., Lops, P., de Gemmis, M., and Semeraro, G. Content-based personalization services integrating folksonomies. In Di Noia, T. and Buccafurri, F., editors, *E-Commerce and Web Technologies*, volume 5692 of *Lecture Notes in Computer Science*, pages 217–228. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03963-8. doi: 10.1007/978-3-642-03964-5_21. URL http://dx.doi.org/10.1007/978-3-642-03964-5_21.
- Nest. Nest thermostat, 2011. URL <https://nest.com/thermostat/meet-nest-thermostat/>. Retrieved: 2016-01-11.
- OASIS. Web Services Dynamic Discovery (WS-Discovery) Version 1.1, July 2009. URL <http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html>. Retrieved: 2017-08-03.
- Parra, J., Hossain, M. A., Uribarren, A., Jacob, E., and El Saddik, A. Flexible smart home architecture using device profile for web services: A peer-to-peer approach. *International Journal of Smart Home*, 3(2):39–56, 2009.
- Pering, T., Anokwa, Y., and Want, R. Gesture connect: facilitating tangible interaction with a flick of the wrist. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 259–262. ACM, 2007.
- Philips. What is hue, 2012. URL <http://www2.meethue.com/en-us/about-hue/what-hue-is/>. Philips Hue introduction. Retrieved: 2016-01-11.
- Pohjanheimo, L., Keränen, H., and Ailisto, H. Implementing touchme paradigm with a mobile phone. In *Proceedings of the 2005 Joint Conference on Smart Objects and Ambient Intelligence: Innovative Context-aware Services: Usages and Technologies*, sOc-EUSAI '05, pages 87–92, New York, NY, USA, 2005. ACM. ISBN 1-59593-304-2. doi: 10.1145/1107548.1107576. URL <http://doi.acm.org/10.1145/1107548.1107576>.
- Qivicon. Qivicon, Living in a Smart Home, September 2011. URL <https://www.qivicon.com/en/>. Retrieved: 2016-02-08.
- Riecken, D. Introduction: personalized views of personalization. *Communications of the ACM*, 43(8):26–28, 2000.

- Rogers, Y. Moving on from weiser’s vision of calm computing: Engaging ubicomp experiences. In *UbiComp 2006: Ubiquitous Computing*, pages 404–421. Springer, 2006.
- Rukzio, E., Broll, G., Leichtenstern, K., and Schmidt, A. Mobile interaction with the real world: An evaluation and comparison of physical mobile interaction techniques. In *Ambient Intelligence*, pages 1–18. Springer, 2007.
- Sadiku, M., Musa, S. M., and Nelatury, S. Internet of things: an introduction. *International Journal of Engineering Research and Advanced Technology*, 2(3): 39–43, 2016.
- Samsung. SmartThings, 2014. URL <https://www.smarthings.com/>. Retrieved: 2016-01-25.
- Scott, M. L. *Programming Language Pragmatics*. Elsevier Science, San Francisco, CA, USA, third edition, 2009. ISBN 9780080922997.
- Seiger, R., Keller, C., Niebling, F., and Schlegel, T. Modelling complex and flexible processes for smart cyber-physical environments. *Journal of Computational Science*, 2014.
- Strohbach, M., Gellersen, H.-W., Kortuem, G., and Kray, C. *UbiComp 2004: Ubiquitous Computing: 6th International Conference, Nottingham, UK, September 7-10, 2004. Proceedings*, chapter Cooperative Artefacts: Assessing Real World Situations with Embedded Technology, pages 250–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-30119-6. doi: 10.1007/978-3-540-30119-6_15. URL http://dx.doi.org/10.1007/978-3-540-30119-6_15.
- Texas Instruments. LM35 Precision Centigrade Temperature Sensors Datasheet, August 2016. URL <http://www.ti.com/lit/ds/symlink/lm35.pdf>. Retrieved: 2016-08-26.
- Thompson, C. Smart devices and soft controllers. *Internet Computing, IEEE*, 9(1): 82–85, Jan 2005. ISSN 1089-7801. doi: 10.1109/MIC.2005.22.
- Tschafenig, H., Arkko, J., Thaler, D., and McPherson, D. R. Architectural Considerations in Smart Object Networking. IAB RFC 7452, March 2015. URL <https://rfc-editor.org/rfc/rfc7452.txt>.
- Vermesan, O. and Friess, P. *Internet of Things—From Research and Innovation to Market Deployment*. Aalborg: River Publishers, 2014. ISBN 978-8793102941.

- Wang, F., Hu, L., Zhou, J., Wu, Y., Hu, J., and Zhao, K. Software toolkits: Practical aspects of the internet of things—a survey. *International Journal of Distributed Sensor Networks*, 2015, 2015.
- Want, R., Fishkin, K. P., Gujar, A., and Harrison, B. L. Bridging physical and virtual worlds with electronic tags. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 370–377. ACM, 1999.
- Weiser, M. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.
- Weiser, M. and Brown, J. S. Designing calm technology. *PowerGrid Journal*, 1(1): 75–85, 1996.
- Weiser, M., Gold, R., and Brown, J. S. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM systems journal*, 38(4):693–696, 1999.
- Wikipedia. List of sensors, August 2016. URL https://en.wikipedia.org/wiki/List_of_sensors. Retrieved: 2016-08-25.
- Youngblood, G. M., Heierman, E. O., Holder, L. B., and Cook, D. J. Automation intelligence for the smart environment. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2005.