Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 109C (2017) 440–447

Procedia Computer Science

www.elsevier.com/locate/procedia

The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017)

An Object-Oriented Model for Object Orchestration in Smart Environments

Luca Bergesio*, Ana M. Bernardos, José R. Casar

Universidad Politécnica de Madrid, Telecommunications School, Av. Complutense 30, 28040 Madrid, Spain

Abstract

Nowadays, the heterogeneity of interconnected things and communication technologies creates several small worlds composed of a single object and a smartphone. For each object, the user needs to download a specific application, search and connect the device. The result is a waste of valuable resources: several objects are able to communicate with the smartphone, but they cannot directly interact among them. In this paper, we propose a model that can be used to define a set of standard interfaces suitable for every smart object. Devices that adhere to the same model can be easily controlled and placed in relation among them, creating multi-object behaviors for a smart space. The smartphone is still a control center, but with a single application it is possible to control and personalize spaces in a holistic way, instead of using the traditional one-to-one approach. Moreover, personalization should be portable: it is desirable that a behavior works in as many smart spaces as possible, at least in a similar way as it does in the environment in which it was configured, freeing the user from the tedious task of adapting it manually every time s/he goes to another space. A portable personalization extends the *bring your own device* paradigm to a new *"bring your own space"* paradigm. The model is inspired in the object-oriented programming, reinterpreting features such as inheritance and polymorphism to the real world, so it is possible to provide a software system able to adapt existing behaviors to new spaces. The use of the model is exemplified in the paper with two examples of smart spaces.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: Smart object; ubiquitous computing; modeling; object-oriented programming; ECA; smartphone; smart spaces; IoT

1. Introduction

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. In fact, their strengths do not lie in their hardware, but in the capabilities to manage interactions among them and in the resulting orchestrated behavior. With the rapid spread of smartphones, often used as a "control center", smart devices have also become popular, with an expected diffusion of 50 billion of things by 2020. Their increasing

* Corresponding author. Tel.: +34 91 453 35 35 ; fax: +34 91 336 58 76.
E-mail address: luca.bergesio@grpss.ssr.upm.es

Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2017.05.415

number will progressively make harder for the users to be aware of all the objects' capabilities and their possible interactions, limiting the ability of the users to control them and to create personalized smart spaces.
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 stated in [1]. Personalization processes usually initiate with some kind of customization input, in which the user configures the system according to his preferences. How to evolve from customizable to personalizable smart spaces is an open challenge. Additionally, it is also open how to make portable the achieved personalization, i.e. how to enable a user to move through different smart spaces preserving and adapting the personalization knowledge to each environment.

In this paper, we propose an object-oriented model able to describe any kind of smart object, and that can be used by an autonomous system to help users to personalize a smart space and to provide an automatic adaptation of a "personalization", when moving to another environment.

2. State of the Art

The concept of smart space evolves from the definition of ubiquitous computing: it is *"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"* [2]. This definition shows spaces composed of objects that work together to improve users' live/experience while they are in that environment. How the space should act to achieve those improvements opens the challenge of personalization in smart environments. Objects should work in a certain coordinated way, decided by the users or by an external system able to learn from their preferences or their behavior.

Personalization is defined by Blom [3] as a process of changing the functionality, interface information content, or distinctiveness of a system to increase its personal relevance to an individual. In smart environments, personalization consists in adapting a space to its dwellers. It can be done by the dwellers themselves (explicit personalization), directly by the environment that configures itself according to the dwellers' preferences (implicit personalization) or according to previous dwellers' preferences (predictive personalization). Nowadays, explicit personalization is enabled through a wide range of smart objects (especially for home automation) that work in cooperation with smartphones, through frameworks that facilitate the communication (e.g. Apple HomeKit, Google Brillo, etc.). Following this approach, we proposed our solution: MECCANO [4], where the smartphone discovers objects in a smart space, it acts as a mediator during the configuration, using the ECA paradigm, and finally it orchestrates the interactions among objects. Research works often propose ad-hoc implementations instead of using commercial frameworks. In [5] are gathered several works that apply artificial intelligence techniques to smart homes.

The potentiality of mobile devices to interact with the environment has been considered both in literature and in commercial applications from some years now. In the late nineties, Beigl [6] suggested a first interaction model between mobile phones and smart objects, using SMS. In the smartphone era, [7], [8] and [9] use the smartphone sensors, such as NFC reader, camera and accelerometers to establish a connection with another device and then being able to interact with it. The frameworks cited before (HomeKit, Brillo, etc.) are nowadays integrated with the two main mobile operating systems and they permit to control and interact with a lot of commercial smart devices. Standalone applications are also available. One of the most popular is IFTTT [10] that uses chains of simple conditional statement to create services that mashup capabilities from smart objects. We proposed a similar approach in [4], using the event-condition-action (ECA) paradigm. Both [10] and [4] are not only a one-on-one solution where the smartphone directly controls another device, but they permit to create more complex behaviors that involve more than one smart object.

The increasing number of smart objects has highlighted the need of modeling these devices to be able to synergize their behaviors, to control them through a single framework and to define a standard process to design new ones. In [11], authors propose a model focused on the objects' (sensing) capabilities, using UML class diagrams to describe the sensors. In [12], the internal components and processes of a smart object are modeled to propose a manner to 'smartify' any device and to provide communication among various objects. [13] extends [12], aggregating the smart devices in an intelligent environment that supports domestic tasks. Likewise, in [14], authors define an architecture of objects that produce a knowledge base, used by a central node to take decisions and to control the actuators.

3. Workflow

In this paper, we consider a mobile-orchestrated architecture, where the smartphone is the key device. It is used by the user to configure the smart objects and to coordinate their tasks. A group of configured tasks is called scene and the execution of a scene produces a particular coordinated behavior of several objects in a smart space. Scene are configured using the ECA paradigm. Every time that a new smart object is added to an environment, it passes through

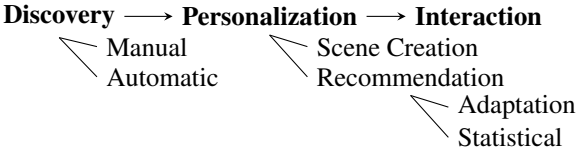

Fig. 1: System workflow.

three different phases before being able to operate. The three phases are discovery, personalization and interaction (see Fig. 1).

During discovery, the smartphone becomes aware of the devices that surround it and of their capabilities. This happens thanks to the download of a module, a piece of software that contains a driver and the object's capabilities (see Fig. 2). Discovery may be manual, when the user performs some kind of interaction with the object (e.g. NFC or QR reading) to obtain a reference to download the module, or automatic if the process does not need any user intervention (e.g. object has some discovery protocol such as UPnP, DLNA, mDNS, etc.).

Once the user has discovered the objects, he can create a personalized behavior, mashing up the capabilities of different devices and respecting the ECA paradigm. ECA paradigm permits to describe behaviors in the form: *WHEN an event occurs, IF a condition is satisfied, THEN do an action* (e.g. WHEN I open the door, IF it is night, THEN switch on the light), where the cardinality of the events for a given scene is 1, of the conditions is 0..* and of the actions is 1..*. For conditions and actions, which allow more than one proposition, only AND and NOT connectives are accepted. In this stage, the user personalizes the smart space to his preferences. As for discovery, personalization may be performed by the user (scene creation) or by an automatic system (recommendation). Recommendation can be achieved through two different approaches. The first one is the adaptation of an existing scene, created by the user, to a new smart space. The second approach uses statistical data about other shared scenes and the user's preferences to recommend an existing scene modified with the user's preferences.
During the personalization phase, the objects that need a particular configuration to work in a scene are set up (e.g. a thermometer, that triggers an event when the temperature reaches 20 °C, must know that value of 20 °C before the execution of a scene to be able to generate the event).

When the scene is configured and the involved objects are ready, the behavior can be performed. The execution is always controlled in a transparent way to the user. It receives the trigger event, then it decides when to check the conditions, if any, and finally it enables the actuators to do the action(s). Since the smartphone is in charge of coordinating the objects to execute a scene, the smartphone is called orchestrator and this workflow phase is called interaction, because it interacts with all the devices involved in the scene.

4. Object-Oriented Model

The workflow points out the role of the smart objects and the smartphone, and the need of a common interface between them, that allows to perform discovery, personalization and interaction. The definition of the common interface passes through the description of the smart objects and their capabilities with a model.

Let us consider a physical object. It exists for one or more specific aims, which are the result that people perceive using or observing the object. It provides some 'passive' functionalities that a user exploits for his needs. They are 'passive' because eventually is the user who uses them (e.g. scissors can cut paper, but the user puts to use them to cut). A smart object is an enhanced version of the corresponding physical object. It has a communication interface that enables the direct interaction with other objects and some electronic components that permit an autonomous behavior. A smart object usually maintains the same service goals, and thus the capabilities of the corresponding non-smart

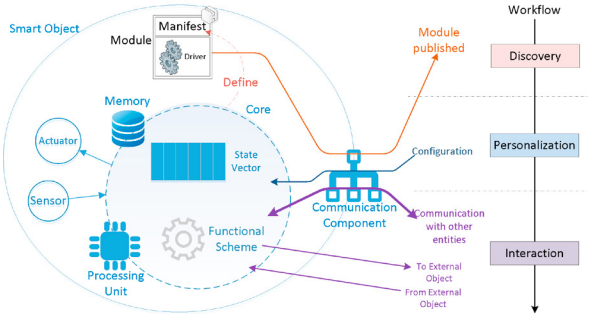

Fig. 2: Model of a generic smart object.

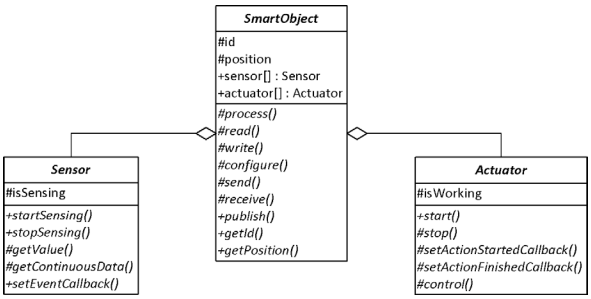

Fig. 3: Generic smart object described with a class diagram.

version. We assume that, an external entity may use/call the capabilities and then control the object behavior. When an entity calls a capability of a smart device, it internally processes the call. To do that, the object uses a set of processing mechanisms and a set of states. Then the smart object converts an order into a real behavior. Reusing some terms proposed in [12], 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 functional scheme is a set of rules that control the transitions between the states. 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, some are public while others are internal to 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 an object's module in a section called *manifest*. In most cases, capabilities depend on object's components that interact with the environment (sensors and actuators). The aforementioned features require some hardware components:

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 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.

Fig. 2 shows all the components and their role along the workflow. Observing this diagram and considering the steps of the workflow that lead to create and execute a scene, we can see several similarities with the object-oriented programming. An object has data (state vector) and code (functional scheme), a scene is an algorithm that contains a declarative section (discovery) and a sequence of operations and calls to methods (capabilities), the execution is performed invoking those methods on objects, etc.

The generic model proposed in Fig. 2 can be formalized and converted in the class diagram shown in Fig. 3. The object's core is represented by the abstract class `SmartObject`, likewise sensors and actuators are modeled with abstract classes. A smart object is an aggregation of sensors and actuators. Attributes represent the state vector and

methods represent the functional scheme. A physical smart object is modeled inheriting from the `SmartObject` class. Physical sensors and actuators need another intermediate abstract class that typifies them (see Fig. 4 and 5). The intermediate class specifies the fields (if applicable) that its subclasses should be able to handle/measure (e.g. in Fig. 4 `Thermal` class has a temperature field, hence physical thermal sensors, that inherit from it, should be able to handle/measure a temperature value). With the intermediate classes, we can classify objects according to their sensors and actuators. This feature is used to help the user during the personalization phase (see next section). In the class diagram in Fig. 3, some fields and methods are reported. Fields represent the basic information and the states that the objects and their sensors and actuators should have. Their presence is not mandatory, therefore a subclass can use or ignore them, for this reason they have a protected modifier (except `sensor[]` and `actuator[]` arrays). In `SmartObject` class, `id` stores a unique identifier for an object (e.g. MAC address, UUID, serial number, etc.), and `position` stores its location, actual location if the object has some location system or a preconfigured one set by the user (e.g. the television in the living room). The position can be useful to identify an object during personalization, especially in those cases where there are identical devices and the identifier cannot be easily recognized by the users. Sensors and actuators are stored in public arrays to permit to external entities to use their capabilities. Classes that represent them have two fields that describe whether the component is working or not (`isSensing` for sensors and `isWorking` for actuators).

As for the fields, classes have some methods with public or protected modifiers. `SmartObject` has three public methods: `publish()`, that represents the capability of the object of publishing its module during discovery, `getId()` that provides the object's identifier and `getPosition()` that returns its location. The other protected methods are the functional scheme: internal capabilities that the object employs to control its internal components. `process()` represents the capability to use the processing unit, `read()` and `write()` to use the memory, `send()` and `receive()` to use the network interface, and `configure()` is the capability of setting up its states and components.

Sensor and actuator class have five methods related to their nature. Sensors acquire data from the environment, therefore they have capabilities that permit to read a physical quantity (`getValue()`), to constantly control a continuous signal (`getContinuousData()`) and to generate an event related to a physical parameter (`setEventCallback()`). Since ECA paradigm needs an event to trigger a scene, the last capability is the most important one for the sensors, for this reason it has a public modifier. The generation of an event is modeled as a callback that calls a function on the orchestrator. `getValue()` and `getContinuousData()` are used to check conditions. `startSensing()` and `stopSensing()` are also public and are used to enable or disable the sensor. Actuators modify the environment, therefore their main capability is represented by the public method `start()` that simply activates an actuator. Several actuators can only be enabled: their automatically stop when they finish their task, for this reason `stop()` is protected. `setActionStartedCallback()` and `setActionFinishedCallback()` are used to notify when an action has begun and when an action has finished respectively. They can be also used as events to trigger other scenes. Finally, `control()` permits a real-time control of the actuator. All methods are virtual, the real objects and components must provide their implementation or hide them using the access modifiers. They may also add their own methods.

In the model are reported method signatures, but they do not have either parameters and 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. In sensors and actuators, only `getValue()` and `getContinuousData()` have a return type, while only `control()` has an input parameter. Sensors' `setEventCallback()` takes a parameter that is a conditional clause regarding the physical quantity that a specific kind of sensor is able to control (e.g. temperature < 25 °C). When the clause becomes true, the event is triggered. The callback function is defined in the orchestrator and it is always the same. An identifier for the event is passed to the callback function in order to distinguish it. In the Fig. 4 and 5, we represent 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.

5. Validation

A scene is an algorithm where objects and their capabilities are combined to produce a behavior suitable for the user. It can be written using a programming language and described with UML.

For the validation, we will use the following scene: *WHEN the main door is opened, THEN turn the light on AND*

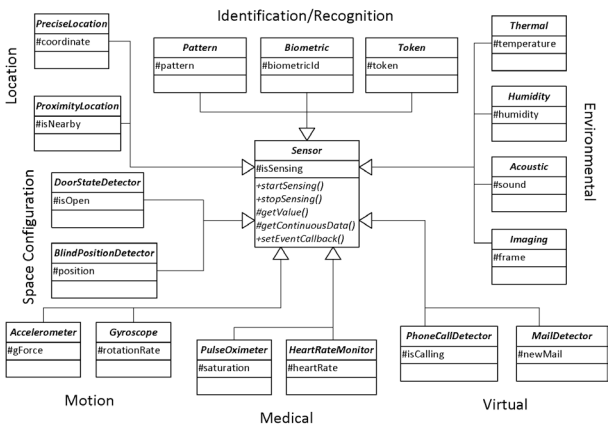

Fig. 4: Class diagram for sensors.

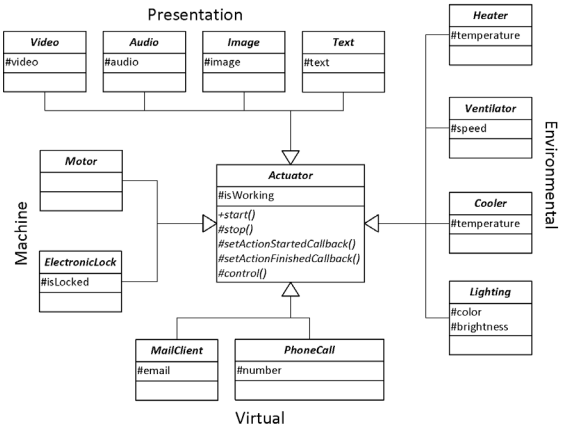

Fig. 5: Class diagram for actuators.

set the color to green. Let us suppose that the user is in a smart space. He discovers two objects: the main door and a smart LED bulb. He receives the modules for both objects and then he creates the scene. A module contains a list of capabilities, possibly with parameters, a parent class definition and a list of inheritance relations between the components and the object. The door is able to generate an event when opened or closed, and this capability is performed by a magnetic sensor, which inherits from a `DoorStateDetector` abstract class, which in turn derives from `Sensor` (`MagneticSensor` ▷ *DoorStateDetector* ▷ *Sensor*). The LED bulb is able to switch on and off using its LED, that is an actuator (`LEDLight` ▷ *Lightning* ▷ *Actuator*). The methods from the door and from the lamp are implementations of the abstract methods included in the model. Modules are a sort of libraries that contains the elements to build a scene. The pseudo code of the scene proposed above is shown in Listing 1.

Listing 1: Pseudo code of the scene used for the validation.

```
SmartDoor mainDoor = new SmartDoor("mainDoor");
SmartBulb smartBbulb = new SmartBulb("light01");
ON
  mainDoor.sensor[MagneticSensor].setEventCallback(isOpen == true);
THEN
  smartBbulb.actuator[LEDLight].start();
  smartBbulb.actuator[LEDLight].control(color = "green");
```

The code is the result of the personalization phase of the workflow. In can be manually created by the user, using a mobile application (e.g. MECCANO[4]), or automatically, created by a recommender system. Mobile application can use the model to assist the user during the manual creation of a scene. In the workflow, we distinguish two scenarios: a statistical analysis of shared scenes and the adaptation of an existing scene to a new space. In this paper, we only consider the second case. Adaptation of the proposed scene means that if I go to a new building, where there are a smart door and a smart lamp, the scene I created on my smartphone will be automatically reconfigured to produce the same behavior using the door and the lamp available in the building. This process ensures the portability of the scenes. The model has been created by observing a generic physical object and converting its components and features into a class diagram. Hence, a scene has been described employing the physical objects as they were virtual objects, following and reinterpreting the OOP concepts. The idea behind the adaptation is to apply the inheritance and polymorphism concepts and their implementations in the compilers to create a system, that works in a similar way, and that is able to exchange objects in a scene. Inheritance and polymorphism in modern OO compilers follow the Liskov substitution principle (LSP): *"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 with other ones, respecting the relations described with the model, and obtaining the same behavior. The implementation of the substitution process passes through two phases: a generalization where physical sensors and actuators are exchanged with their abstract parent class, and an adaptation phase where a new physical component that inherits from the same parent class is inserted into the scene.

Thereafter, the user goes to another space where there are a smart door and a floor lamp. After discovering the objects, his smartphone tries to adapt the scene employing the new objects. Using the model, it generates a new "abstract" scene with the superclasses from which derive the original sensors and actuators. Generalization process of the scene is shown in Fig. 6. Sensors and actuators are substituted by their abstract parents. Removing sensors and actuators, also the original smart objects, to which they belonged, are removed. The scene is then described in terms of abstract sensors and actuators (Listing 2). The generalized scene is something like: *WHEN a door state detector recognizes an opening, THEN switch on a light AND set the color to green*. In this scene, no objects are mentioned, only components (in Fig. 6, objects are removed and the scene becomes an abstract class).

Listing 2: Pseudo code of the generalized scene.

```
ON
  SmartObject.sensor[DoorStateDetector].setEventCallback(isOpen == true);
THEN
  SmartObject.actuator[Lightning].start();
  SmartObject.actuator[Lightning].control(color = "green");
```

It loses importance to know who actually perform a capability: a car able to detect when the trunk is opened and with a multicolor light can perform the scene. A door with an embedded LED strip or a window, a smartphone's flash and a multicolor bed light can do the same. Then the recommender searches among the objects in the new space, whether some of them have components that derive from the classes used in the generalized scene. If the minimum number of objects required to fulfill the ECA cardinality is found, then the scene can work in the new space, at least in a similar way. Some optional capabilities, that belong to a specific object, may not be present in the abstract classes in the model (e.g. a LED bulb with blinking mode). If one of these capabilities is used in a scene, the substitution of the object is possible only with an identical one (i.e. the same LED bulb with blinking mode). On the other hand, the opposite situation is also possible. Abstract classes and capabilities included in the model are thought to be a set of basic capabilities and properties, but it is possible to find components with less capabilities than those defined in the parent class. In the model, this case is possible due to the use of the protected modifier: concrete classes may implement and publish a method, by increasing its visibility, or they may ignore it. In our scene, LED light can change color, but there are also monochromatic LED lights that do not implement `control(color)`.
Let us suppose that in the new space, the user has discovered the door, which is an identical door as the previous one, and a floor lamp. The orchestrator has a generalized scene, now it tries to add the new objects. The door is the identical object, it only has a different identifier. The floor lamp is different: it is a lighting actuator, but it has a monochromatic bulb, thus it cannot change color. The new scene will have the pseudo code reported in Listing 3.

Listing 3: Pseudo code of the adapted scene.

```
SmartDoor livingRoomDoor = new SmartDoor("door02");
FloorLamp floorLamp = new FloorLamp("light02");
ON
  livingRoomDoor.sensor[MagneticSensor].setEventCallback(isOpen == true);
THEN
  floorLamp.actuator[Bulb].start();
```

The bulb of the floor lamp can only be switched on and off, it does not provide any method to control either the color or the brightness. In this case, the color of the light in the new room cannot be changed, the adapted scene is still valid because it respects the ECA cardinality, but it will work in a reduced way. A hypothetical light, that can change its color, but that cannot be switched on or off, may also be used to adapt the scene. In this example, since the cardinality is 1-0-2, the most difficult object to exchange is the door sensor. In general, the event is the most difficult capability to substitute due to its fixed cardinality.
Model also establishes a hierarchy of what can be exchanged. Observing the subclasses of `Sensor` and `Actuator` we can see that the most important capabilities are the public ones. They represent the basic feature that a kind of sensor and actuator should have. In the second place, there are the protected methods that represent an optional feature. Finally, the value of the arguments passed to the objects. Considering the light changing, we can assign a distance value between the original scene and the adapted scene. The distance is 0, if the light can be switched on and set to green; it is 1, if it can be switched on and the color can be changed, but not to green; 2, if it can be only switched on; 3 if it can be only set to green; 4, if the color can be changed, but it cannot be changed to green. Distance value may be used to implement an AI system (e.g. case-based reasoning) that recommends scenes to the users.

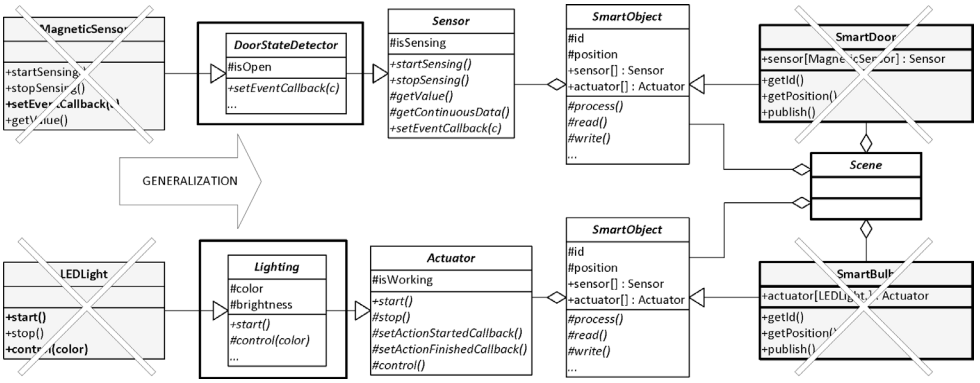

Fig. 6: Generalization of the scene.

6. Conclusion

In this paper, a model to describe smart objects is proposed. 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 is an object-oriented model, smart objects are represented as aggregations of 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 developers should follow increase the level of interaction among the devices. Applying inheritance and polymorphism to the real world, we finally propose a solution to adapt a scene to a new space, exchanging the involved objects, but maintaining the same global behaviors.

Acknowledgments

This work has been supported by the Spanish Ministry of Economy and Competitiveness under grant TEC2014-55146-R and by the Technical University of Madrid under grant RP150955017.

References

1. D. Riecken, Introduction: personalized views of personalization, Communications of the ACM 43 (8) (2000) 26–28.
2. M. Weiser, R. Gold, J. S. Brown, The origins of ubiquitous computing research at PARC in the late 1980s, IBM systems journal 38 (4) (1999) 693–696.
3. J. Blom, Personalization: a taxonomy, in: CHI'00 extended abstracts on Human factors in computing systems, ACM, 2000, pp. 313–314.
4. A. M. Bernardos, L. Bergesio, J. Iglesias, J. R. Casar, MECCANO: a mobile-enabled configuration framework to coordinate and augment networks of smart objects., J. UCS 19 (17) (2013) 2503–2525.
5. J. C. Augusto, C. D. Nugent, Designing Smart Homes: The Role of Artificial Intelligence, Vol. 4008, Springer, 2006.
6. M. Beigl, Point & click-interaction in smart environments, in: H.-W. Gellersen (Ed.), Handheld and Ubiquitous Computing, Vol. 1707 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 311–313.
7. L. Pohjanheimo, H. Keränen, H. Ailisto, Implementing touchme paradigm with a mobile phone, in: Proceedings of Joint sOc-EUSAI, ACM, New York, NY, USA, 2005, pp. 87–92.
8. E. Rukzio, G. Broll, K. Leichtenstern, A. Schmidt, Mobile interaction with the real world: An evaluation and comparison of physical mobile interaction techniques, in: Ambient Intelligence, Springer, 2007, pp. 1–18.
9. G. Broll, E. Rukzio, M. Paolucci, M. Wagner, A. Schmidt, H. Hussmann, Perci: Pervasive service interaction with the internet of things, Internet Computing, IEEE 13 (6) (2009) 74–81. doi:10.1109/MIC.2009.120.
10. IFTTT, Connect the apps you love, URL: `https://ifttt.com/` (2011).
11. A. Fernández-Montes, J. Ortega, J. Sánchez-Venzalá, L. González-Abril, Software reference architecture for smart environments: Perception, Computer Standards & Interfaces 36 (6) (2014) 928–940.
12. C. Goumopoulos, A. Kameas, Smart objects as components of ubicomp applications, International Journal of Multimedia and Ubiquitous Engineering 4 (3) (2009) 1–20.
13. C. Goumopoulos, A. Kameas, Ambient ecologies in smart homes, The Computer Journal 52 (8) (2009) 922–937.
14. M. Strohbach, H.-W. Gellersen, G. Kortuem, C. Kray, Cooperative artefacts: Assessing real world situations with embedded technology, in: International Conference on Ubiquitous Computing, Springer, 2004, pp. 250–267.