An Object-Oriented Model for Smart Devices in Internet of Things

Boris Ulitin, Eduard Babkin
National Research University Higher School of Economics
Nizhny Novgorod, Russia
bulitin, eababkin@hse.ru

Abstract—**The heterogeneity of interconnected devices and communication technologies in the Internet of Things (IoT) domain courses the problem of impossibility to synchronize several different objects in one scheme of the smart space effectively and automatically. Such diversity of technologies results in the need to have special skills in order to reach once created behavior for the smart space. In addition, there are many platforms, which allows to interconnect different devices, but only if they fulfill their protocols. All these lead to the idea, that a new, unified approach to the smart devices representation is needed, which allows to represent objects in a compact form by a platform-independent way. In this article we propose an object-oriented model for representation of the smart devices and demonstrate its efficiency by the simple case of smart space scenes adaptation.**

I. Introduction

Smart objects are electronic devices, which allow to work interactively and autonomously with other objects, using network interfaces. The main strength of such smart devices relies not on their hardware, but on the capabilities to manage interactions among them, that results in orchestrated behavior of the whole smart space.

The concept of smart spaces originates from the definition of ubiquitous computing, which is given in [4] as “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”. This definition shows, that smart space can be represented as a set of interconnected objects (devices), using which users can improve their life and/or experience.

Taking into account the widespread expansion of smartphones, which can be used as a “command center”, smart devices also seems to become popular with an expected diffusion of ~50 billion of things by 2020 [1]. Such a significant increase in the number of devices creates difficulties for users to operate all possible features and complicates the organization of orchestrated work within the united smart space. As a result, there is no opportunity to create personalized smart spaces effectively because of their complexity.

On the other hand, the trend in computing is moving to cheaper, smaller and faster devices in general. However, it comes with its own challenges. First of all, the dominant mode of communication in IoT is wireless and there are already diverse variety of such protocols [1]. In addition to protocol differences, there are platform and format variations that limit the integration possibility [2]. This differences force system developers to utilize a range of tools and require various skills to program. For instance, interoperability middleware [3] are common solutions.

Furthermore, there are two types of interoperability: technical and application. The first one is interoperability on the device level, when we support this process using different standards and unified technical protocols (eg. IEC ISO/IEC JTC 1/SC 41 [16], CoAP (RFC7252) [18], etc.), while the application interoperability means the ability to represent the device capabilities in a structured manner for using them as scenarios, applicable for different smart spaces. The application interoperability needs the conceptual scheme for device capabilities representation in order to ensure the independence of scenarios from the specific implementation of the smart spaces with the preservation of its personalization. As a result, this type of interoperability cannot be provided by standards, which define more technical aspects of the interoperability, than their conceptual representation.

All these leads us to the idea, that in order to simplify the procedure of personalization of the smart spaces, we have to create the unified representation for every device, which participates in it. Such an object-model for the device will allow not to orchestrate initially every new device in the space, but to adapt it according to its unified representation. As a result, the user will have more opportunities to organize a truly personalized smart space without the need to redefine all connections after adding new devices.

In this paper we propose such an object-oriented model 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. Prior to going into the details, in Section II the limitations and challenges in smart spaces and IoT are reviewed to motivate the importance of proposed approach. Then procedure of the device personalization in the smart space (Section III) and description with some analysis of the object-oriented device model are presented (Section IV). We conclude the article with validation of the proposed approach (Section V), introducing the idea of scene distances (Section VI) and specification of the future researches.

II. Challenges in smart spaces and IoT

Inside any smart space there is some IoT concept. As a result, in order to improve the quality and effectiveness of smart spaces, a quick and simple real-time coordination of all devices in terms of IoT have to be provided. However, realization of such coordination procedure faces the many challenges, existing in IoT.

First of all, there are many types of devices in IoT, with different architectures, which are controlled by different operating systems (or, sometimes, even without them), and which have to be inter-connected through a wide specter of communication protocols. Furthermore, some devices get deployed in remote locations or embedded in physical objects, that makes it difficult (or, at worst, impossible) to bring and configure them. This results in really high dimension of heterogeneity in development of effective IoT-based implementation.

There are some efforts in the area of interoperability by introducing a middleware [3] to hide the underlying platform and protocol variation. However, most of them require special skills in programming languages and device settings [2].

Furthermore, new standards in IoT also allow to unify and simplify the process of interactions between different devises. In particular, IEC ISO/IEC JTC 1/SC 41 [16] describes the recommendations on semantic and network connectivity, and ITU-T Y.2066 [17] contains network and data management requirements. Unfortunately, these standards contain only general recommendations and do not propose any adaptations for a concrete area of IoT application.

There are some attempts to resolve this problem. For example, Constrained application protocol CoAP (RFC7252) [18] and MavHome [19] allow to provide interoperability among smart devices in a smart home using a web-application protocol (similar to HTTP). According to MavHome architecture, there are 4 layers of interoperability: decision, information, communication, physical. Physical layer consists of all the physical objects and its interfaces within the smart home environment. Communication layer is responsible for transferring the information between objects and also to the user. Information layer aggregates the data from sensors and actuators to be used for decision making and analysis. Decision layer extracts the knowledge from the information gathered and also uses the information implicitly provided for making decision of what action needed in what kind of scenario. First three levels are mostly described in ISO and ITU-T standards, while the detailed specification of the fourth layer requires further research. In our article we focus on this layer and try to describe the most natural way to represent the capabilities of devices, needed for IoT scene scenarios activating.

One more global problem in smart spaces and IoT, connected with the decision layer, is a problem of personalization, which means the process of changing functionality, interface information content, or distinctiveness of a system to increase its potential relevance to an individual [5]. In other words, personalization is a process of adapting a space to its dwellers. Based on who initiates this process, three types of personalization can be defined:

1) Explicit personalization: takes place, when the user his/her-self configures the environment;

2) Implicit personalization: takes place, when the environment configures itself according to users' preferences;

3) Predictive personalization: can be provided by both the user and the environment independently according to previous users' preferences.

It is fair to note that now explicit personalization enabled through a wide range of smart objects (especially for "smart home" automation [6]) that work in cooperation with smartphones, using frameworks that facilitate the communication [7] (for example, Apple HomeKit [13], Google Brillo [14], MECCANO [7]). The core approach for these frameworks is the same – a smartphone discovers objects in a smart space and acts as a mediator during the configuration using the event-condition-action (ECA) parading, orchestrating the interactions among objects.

The increasing number of smart objects stresses the need of modeling these devices in unified manner in order to have opportunity to synergize their behavior and to control them using a single framework, which provides a standard process to design new ones. Object-oriented approach can be an effective instrument for achieving this goal. But in order to show it, we have to analyze what the nature of smart devices is and how they are implemented in smart spaces.

III. Lifecycle of a device in the smart space

As it was mentioned before, the smart space is usually a set of devices, which are connected to the smartphone, used as a key device to configure the smart objects and coordinate their tasks by the users. A group of configured tasks is called a scene, execution of which produces a particular case of coordinated behavior of several objects in the smart space.

Usually scenes are configured using the Event-Condition-Action (ECA) paradigm [3]. In general, the ECA paradigm stands on the representation of any behavior as triple (Event, Condition, Action), which can be interpreted as follows: WHEN an **event occurs**, IF a **condition is satisfied**, THEN **do an action**. According to this paradigm, every new smart object, included in the smart space, passes through three stages of the lifecycle before being ready to operate (see Fig 1).

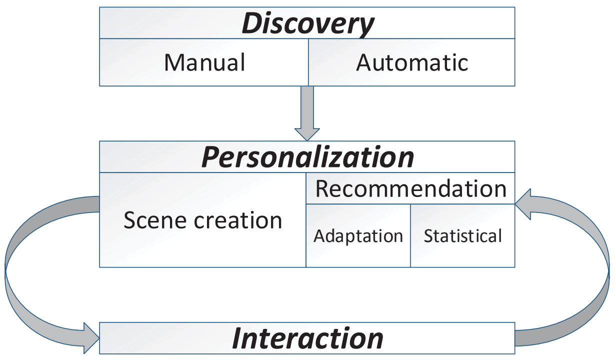

Fig. 1. The lifecycle of the device in the smart space

1) Discovery: during this phase the smartphone identifies, what devices surround it and their capabilities. For this goal different download modules can be used which contain a driver of the object and formal specification of its' capabilities. Discovery can be manual, initiated by the user through interaction with the target object (e.g. NFC or QR reading), or automatic when the process needs no user intervention;

2) Personalization: upon successful identification of the object, the user can create a personalized behavior, mashing up the capabilities of different devices with respect to the ECA paradigm. For the one specific scene the cardinality of Event is 1, of Conditions is 0..* and Actions is 1..*. In other words, in order to describe the scene, we have to identify at least one action for every event with optional reference to some additional conditions. Similar to the Discovery phase, Personalization can also be provided by both the user (scene creation) and an automatic system (recommendation). The last one can be achieved in two different ways: through adaptation of the existing scene to a current space or through statistical analysis of shared scenes with respect to user' preferences to find the most suitable for application to the current scene;

3) Interaction: when the personalization finished, the new behavior can be performed. The control procedures are fulfilled in a transparent way by the user. Through this phase, when the smartphone receives the trigger event, it decides to check the conditions (if there are any) and then enables the actuators to do some actions.

It's obvious that when the actual scheme and interfaces of the device are used, flexibility and opportunity to create an effective orchestrated infrastructure are lost. Much simpler and effective decision consists of defining a single structural scheme for the device representation. Such scheme defines all its components and functional capabilities in one manner. In addition to defining the behavior of different devices in a standardized way, that approach also allows to involve the user in the process of configuration of the smart space without special skills in programming languages and device settings.

IV. THE OBJECT-ORIENTED MODEL FOR SMART DEVICES

A. Device Object Model (DOM)

The concept of DOM is influenced by the Document Object Model [15], which is a platform and a language independent interface exposed so that programs can manipulate its elements. It allows to update and format elements of the document in dynamic way. For example, in [2] there is an example of using Document Object Model for manipulating HTML documents at run-time.

Analogically, DOM represents resources and capabilities of the device in structured manner. Resources can be as hardware resources, and shared resources of the external environment such as sensor reading, private resources of storage, etc. According to this, three components can be allocated in any device in order to build DOM conceptualization (see Fig. 2).

The first group of resources includes resources that are closed from other participants in the environment and defines the hardware limitations of the device: network protocol, memory and battery values, etc. The second group is formed by functional components, which are used by the device to interact with other objects in the space and allows to define the behavior of the device: sensors (allow to monitor changes in the device environment), actuators (to react to the changes, caught by sensors) and events (identify in what cases actuators have to be used). The last group of resources identifies the relevant information about the device (e.g. context, location, environmental conditions, identifier tags, etc.).

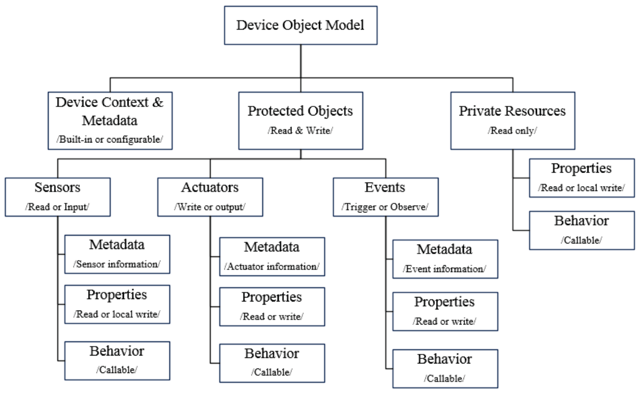

Fig. 2. The concept scheme of Device Object Model

Taking into account the idea of the device personalization, the most useful for us components of the device are represented in the second group of its elements. Namely sensors, actuators and events allow us to set the behavior of the device within the ECA paradigm: Sensors – for Conditions definition, Actuators – for Actions and Events – for itself. If we can formalize the structure of Sensors, Actuators and Events, we can identify the device behavior in unified manner and adapt once created scene to another devices, conditions and spaces.

Furthermore, while the metadata of all devices depend on their hardware, their properties and behavior can be formalized in a platform-independent manner using the object-oriented approach. Such approach replaces every device with its conceptual model, which aggregates properties and functional aspects of the device irrespective of their physical realization.

B. Object-oriented Model

Before analyzing the object-oriented model for smart devices, we should pay attention to the fact, that every smart device is an improved version of its physical object. A physical object exists for some specific goals, which are achieved through observing or using "passive" functionality of such objects by users. These capabilities are "passive" because of impossibility to use them without external user, who calls them.

In contrast to such physical ones, smart objects have the ability to expose their own behavior without any external interference. For they have communication interfaces, which enable direct interactions with other participants of the smart space, smart objects can permit autonomous behavior. It is for this reason that the user's access to the specific capability of the smart device is not the fact of calling the corresponding procedure, but a request to a specific interface that ends in the internal processing the call. To do that, the smart object uses a set of processing mechanisms with a set of states [7], which convert the original request into a real device behavior.

Following Goumopoulos [8], we identify the processing mechanism of the device as a *functional scheme*, and a set of its states as a *state vector*. The second one defines at every specific time moment current conditions (or state) of every device's component (sensors, actuators, etc.), while functional scheme is responsible for transitions between different states, determining the rule for state vector change according to the data, received from device's components. Under these conditions, it can be argued that the functional scheme is the implementation of capabilities, structured in the way of differentiation of the access to them. When a user calls a device's capability, the functional scheme tries to execute this capability and change the state vector of the corresponding device [6]. Capabilities become defined during the discovery phase of the device lifecycle, and the functional scheme is the result of the personalization stage.

In our case, we perceive the component structure of devices not as representation of their technical parts, but as a conceptual model of their capabilities, encapsulated in the respective objects. This view of the smart device is completely consistent with the approach proposed by Sebastian in the [20], where the following levels of IoT architecture are distinguished: Device (contains physical description and implementation), Communication (performs the communication between devices), Services (serves various types of functions for device discovery, modeling, control, etc.), Management, Security and Application (provides necessary modules to control, and monitor various aspects of the device and IoT system in general). In what follows, we pay attention only to the layer of applications and services, as having the greatest importance for personalizing the smart space and organizing its behavior in the form of generalized scenarios and scenes. As a result, when we tell about technical capabilities, we mean not their concrete implementation, but the corresponding service (method), allowing to activate the corresponding capability of the device (or its part).

Obviously, the implementation of the aforementioned features requires appropriate technical capabilities, implemented in the corresponding hardware components, which can be separated into two categories: *core components* and *supporting components*.

Core components enable to define and run scenes in the smart spaces. These components can be conceptualized and adapted from one smart space to another. This group is organized by the next objects:

- **Sensors** – devices, which allow to detect changes (or events) in the smart space. As a rule, two modes of the sensing are available: on demand or continuous. Sensors are objects, responsible for checking conditions, demanding on physical parameters of the space.
- **Actuators** – opposite to sensors, these objects do not react to changes in the space but provide them. Actuators change state vectors of other devices in the space, that is the result of changing in the state vector of the actuator's device. Together with sensors, actuators create a closed space change system: sensor change switchers of the state vector, and actuator respond to these changes.

On the other hand, there are hardware elements, which are responsible for physical implementation of the scene. These elements are supporting, since they do not introduce anything new into the scenario of the scene, but only trace the commands embedded in it on the physical level of the devices. Although conceptualization of such devices is quite difficult, the set of supporting devices includes:

- **Processing module** – is a component, responsible for execution of the functional scheme. Depending of the device it can be CPU, GPU, a microcontroller or a combination of many of them.
- **Storing module** – is used to store the state vector and/or historical data (events, scenes, evolution of the state vector, etc.), configurations, etc.
- **Communication component** – this component allows devices to interact between themselves and can locate the objects within a space during the discovery phase.

Analyzing the aforementioned structure of the device components, we can find opportunity to draw the parallel between it and object-oriented program. In the same way, as an object has data and code, the core components of the device interact with the state vector and the functional scheme. Furthermore, the scene for the smart space can be represented as an algorithm with a declarative section (which is a result of discovery phase) and a sequence of operations/calls to methods, which are performed by supporting components [7]. As a result, we can formalize the generic structure of the device in the class diagram (see Fig. 3).

In this diagram a device is represented by the abstract class *SmartObject*, and its core components (sensors and actuators) by the abstract classes of the same name. As a consequence, the object is represented as an aggregation of its own sensors and actuators. Attributes of classes represent the state vector, and its methods represent the functional scheme. In these circumstances, the real smart object will be modelled as an inheritor of the *SmartObject* class. As for sensors and actuators, they can be typed using a system of heirs, and not just using a single abstract class. Examples of such typification, based on the goal and main tasks of the objects, were provided in more details in [7] and [2].

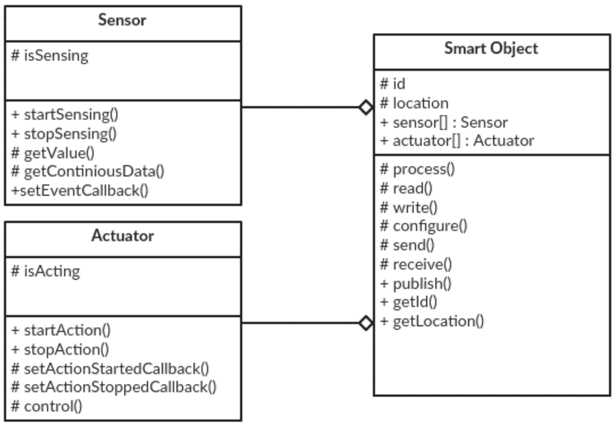

Fig. 3. UML-representation of the object-oriented device model

In the diagram above some fields and methods are also reported, which basically are needed for effective interaction with any device. Their presence is not mandatory and can be ignored by the physical object realization, and therefore they are marked as protected (except sensors and actuators, which represent the core components of the device).

SmartObject class is characterized by the next set of fields:

- *id* – stores a unique identifier of the object (e.g. MAC address, UUID, etc.);
- *location* – stores the actual location of the object, or fixed by the user (e.g. TV in the kitchen, etc.), and can be used as an equivalent for the id in cases, when id is difficult for user recognition;
- *sensor[]* – represents the list of capabilities, provided by the sensors, which the device contains;
- *actuator[]* – represents the list of capabilities, provided by the actuators, which the device contains.

isSensoring field in *Sensor* class and *isActing* filed in *Actuator* class correspondingly identify whether the component is working or not.

As for the fields, classes also have some methods with public or protected modifiers to interact with the devices. *SmartObject* class contains the next set of methods:

- *publish()* – guarantees the opportunity to identify the object and its components during discovery phase;
- *getId()* and *getLocation()*– return the value of the identifier and location fields of the current object correspondingly;
- *process()* – represents the ability to run the process, using the processing module of the object;
- *read()* and *write()* – are used for interactions with the memory module;
- *configure()* – provides the opportunity to set and change the state of the current object;
- *send()* and *receive()* – provide access to the communication component and support interconnections between objects.

Sensor and *Actuator* classes also have their own methods, which simplify the process of their behavior implementation.

Because the main goal of sensors is to catch changes in the environment state, *Sensor* class has two methods: *startSensing()* and *stopSensing()*, which correspondingly initiates and finishes the process of analyzing the environment changes respectively. When the sensing is initiated, we can read a physical indicators of the environment using methods *getValue()* for manual controlling of the indicator value and *getContinuousData()* for continuous monitoring of changes in environmental indicators. After changes are fixed, we have to report this information to actuators and other objects using *setEventCallback()* method, which generate an event related to the environment indicator.

In terms of the ECA paradigm and scene definition, the last method is the most interesting, because it allows to identify the event trigger to the event, which in turn, can be verified by means of *getValue()* and *getContinuousData()* methods.

Analogically, *Actuator* can initiate the process of the environment's changes using *startAction()* and *stopAction()* methods. In both cases actuator has to notify other participants in the environment about provided action. For this goal methods *setActionStartedCallback()* and *setActionStoppedCallback()* are used. And finally the current state and the actions of the actuator can be controlled by the *control()* method.

Being members of abstract classes, all these methods are virtual and have to be implemented or hidden in the real objects. We also do not concretize any return types and parameters for methods in order to do them as generic as possible. A user can specify them during the personalization stage, for example, specific return types for *getValue()* and *getContinuousData()* methods; input parameter for *control()* method and conditional clauses as an input parameter for *set*Callback()* methods.

Such proposed object-oriented model becomes an effective, clear and simple tool to extension of representation for the smart devices. If a user needs to add a new device to the smart space, he/she has only to inherit a corresponding abstract class and define all needed components during its personalization. On the other hand, once created the scheme of the smart space, can be transferred into another physical implementation of the space without need to repeat all stages of the devices' lifecycle. In this case the user can only define, what new devices are physical equivalents to stored abstract components and initiate the similar behavior in a new environment in real-time.

V. VALIDATION

Taking into account that the scene for the smart space is an algorithm over capability of devices, we will represent it using a pseudo-code for the scene and a UML class-diagram for devices' components.

We will use simple scenarios in validation part, because the main application domain for our model is Smart Home, where the number of devices is limited, but which are very heterogeneous because of different goals for their application. In case of large-scale IoT systems the application of scene adaptation by the user seems not the most effective way and, as a consequence, goes beyond the boundaries of this study.

In order to show the effectiveness of the proposed object-oriented model we will use the next scenario: *WHEN the temperature is more THEN 25°C THEN turn the ventilator AND set the speed to 1000 rpm.*

Imagine a user in the smart space. The user discovers two objects: the smart thermometer iTerm and the smart ventilator Smart Fan. The user received modules for both objects, which contain a list of capabilities, a parent class definition and a list of relations between the components and the object, and creates the scene. The thermometer can generate the event when the temperature exceeds the upper limit of 25°C, which is performed by the Thermal sensor, which is the inheritor of the Sensor class (iTermThermalSensor->ThermalSensor->Sensor). In its turn, the ventilator can change its speed using SmartFan Speed actuator (SmartFanActuator->Ventilator->Actuator). Both the actuator of the ventilator and the sensor of the thermometer have implementations of the abstract methods included in the model. The scene in such conditions can be represented in terms of a pseudo-code like in Algorithm 1.

This code is produced during the personalization stage and can be created both manually by the user, or automatically by the recommender system [9]. Once defined, this scene can be used for adaptation in the future smart space. This process can be organized through statistical analysis or adaptation of the scene to new devices. In our work we concentrate on the second one way for adaptation.

Algorithm 1 Pseudo-code of the scene

```
1: SmartThermometer iTerm = new SmartThermometer("iTerm");
2: SmartVentilator smartFan = new SmartVentilator ("smartFan");
3: ON
4:   iTerm.sensor[iTermThermalSensor].setEventCallback (temperature > 25);
5: THEN
6: smartFan.actuator[SmartFanActuator].start();
7: smartFan.actuator[SmartFanActuator].control(speed = 1000);
```

The main idea of such adaptation is to reconfigure the created scene according to capabilities of new devices. Here we also use the principle from the object-oriented programming.

It is assumed that we have already built a conceptual (or, virtual) scheme for the scene and devices, participated in it. And in the stage of personalization we only concretize the abstract classes with its physical implementations. As a result, we use principles of inheritance and polymorphism from the object-oriented approach. Taking into account the fact, that these concepts fulfill the Liskov substitution principle (LSP), we can tell, that in the process of scene adaptation, smart objects, sensors and actuators are exchanged with other ones, saving the relation, described with the abstract model and, as a result, obtaining the same behavior.

According to this, two phases of scene adaptation can be defined: *generalization*, the main idea of which is to generalize the created scene by replacing all physical implementations with its abstract parent classes, and *direct adaptation*, where abstract classes are replaced with its inherited new physical components implementations.

If we apply that approach to our example, at the beginning we should create the scheme for components, participating in created scene (see fig. 4). In this scene we represented all components as UML-classes with corresponding names, fields and methods. We also saved the relations between created classes and its parent abstract classes, which are, in common, the inheritors of the general abstract classes *Sensor*, *Actuator* and *SmartDevice*. In addition to these, the diagram contains the abstract class *Scene*, which contains several smart objects and is a representation for our physical smart space.

Now we start generalization of the created scene. First of all, we rewrite its pseudo-code representation with a new, more abstract version, which is based on abstract classes instead of physical implementations of components. This abstract version of a pseudo-code is represented in Algorithm 2.

In this version of the scene scenario, it loses importance, who actually perform the corresponding capability. It can be any example of Thermometer with the ability to measure the temperature with any example of Ventilator. Furthermore, these devices can be replaced with equivalents from the point of the functionality, for example, an air conditioner in the car and mobile application for measuring temperature.

This means, that this scene can be adopted not only to other devices, but to the new smart environment. And once created behavior can be applied to any device, inherited from the corresponding typified abstract class (a sensor or/and an actuator).

Algorithm 2 Pseudo-code of the abstract scene version

```
1: ON
2:   SmartObject.sensor[ThermalSensor].setEventCallback (temperature > 25);
3: THEN
4: SmartObject.actuator[Ventilator].start();
5: SmartObject.actuator[Ventilator].control(speed=1000);
```

Updated, the generalized diagram, is represented in fig, 4 (the framed part). Here we see only abstract classes, without its physical inheritors. In order to implement this generalized scene to the new smart space, user have to identify only, what new physical objects are inheritors of the generalized scene abstract classes and after it permit corresponding behavior embedded inside the generalized scene.

What is more important, the user can not only build in a new environment a mirror image of the previously created scene but adopt it to the new devices and its restrictions. For example, if a new specific Ventilator does not support the speed setting capability, user can delete corresponding command from the scene (see Algorithm 3) and apply the updated one to the physical implementation.

Algorithm 3 Pseudo-code of the updated scene version

```
1: ON
2:   SmartObject.sensor[ThermalSensor].setEventCallback (temperature > 25);
3: THEN
4: SmartObject.actuator[Ventilator].start();
```

Such adaptation can be provided in semi-automatic way, using the idea of distance between the generalized scene and its physical implementation, describing in the next section.

VI. Automation of the Adaptation Using Distance Between the Scenes

Before introduction the concept of the distance definition, we have to identify the formal representation of the scene for the smart space.

From the formal point of view, any scene can be represented as a set of sensors and actuators, which capabilities are embedded in it. It means, that *Scene = (S,A)*, where *A*ϵ*Obj* and *S*ϵ*Obj*, and *Obj = {obj$_i$: obj$_i$ = (T, F,M), where T is a type of the obj$_i$, F = {attr$_j$}, j = 1..N and M = {method$_k$}, k = 1..S}*.

As a result, the scene can be described as a set of its sensors and actuators, which, in its turn, are defined using their type (which allows to define the opportunity to replace one object with another), attributes and methods. Such definition has advantages, because allows to formalize the process of the automated scene adaptation and evaluate its possibility, identifying the need of user's interventions.

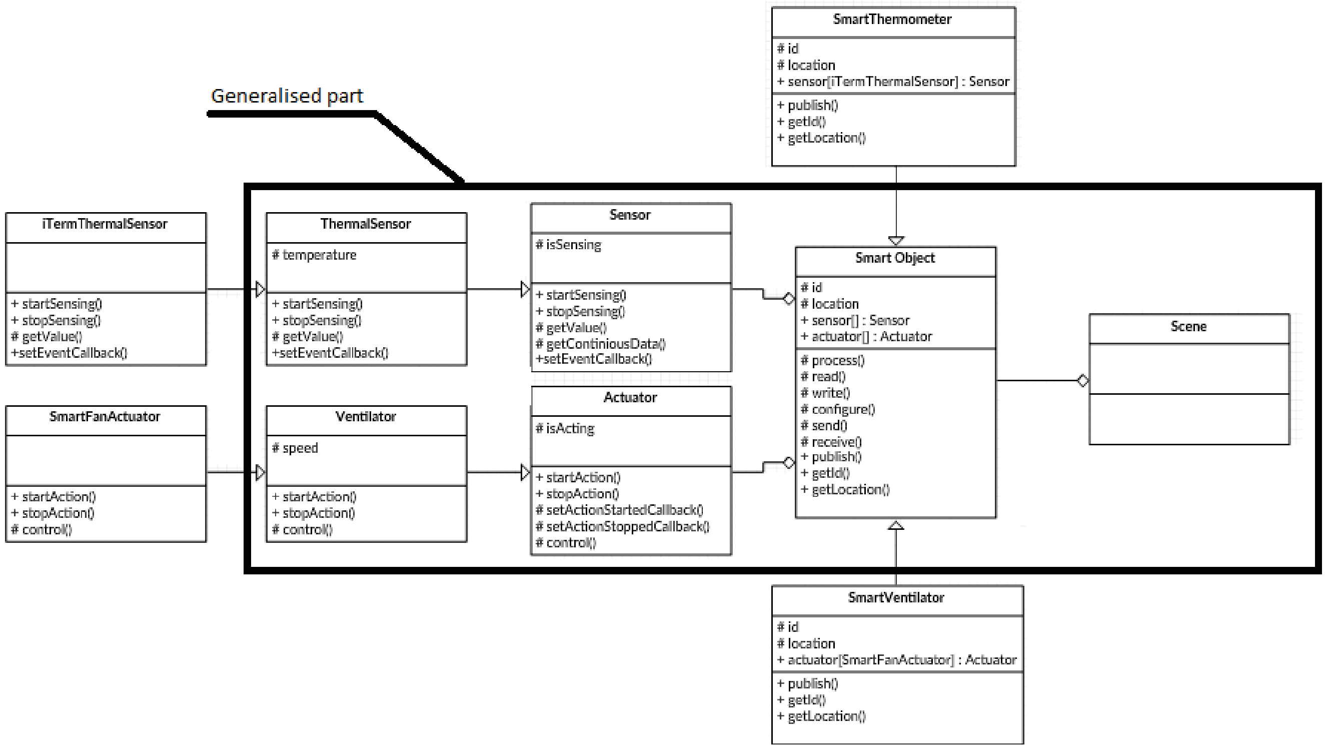

Fig. 4. UML-representation of the created scene and its abstract generalization

In what follows, we will also need the next statement. Two objects (sensors or actuators) are equivalent, if their types are equals: *Obj$_1$ = Obj$_2$ if T$_1$ = T$_2$*.

Now, using these statements, we can provide the definitions of the compatible and adaptable scenes.

Definition 1: Two scenes *Scene$_1$* and *Scene$_2$* are compatible iff $S_1 \cap S_2 = S_1$ and incompatible otherwise.

This definition states, that one scene can be transformed into another in the case, when they have an equivalent set of sensors. It is logically correct from the position of the ECA paradigm, because if the original scene has a sensor which is absent in the target scene, it is impossible to check all conditions for execution of the scene scenario. If two scenes are compatible, then we can try to adopt the first one (original) to the second (target) without user's interventions, because can check all the conditions of the original scene in the new smart space.

We can reformulate this Definition 1 using the idea of distance between scenes. In order to achieve this, the next definition is fixed.

Definition 2: Sensor distance between two scenes *Scene$_1$* and *Scene$_2$* is a value: $Dim_S(Scene_1, Scene_2) = |S_1| - |S_1 \cap S_2|$.

Using this, we can reformulate the Definition 1 as follows: Two scenes *Scene$_1$* and *Scene$_2$* are compatible iff $Dim_S(Scene_1, Scene_2) = 0$ and incompatible otherwise.

Now, when we introduced the concept of compatibility of two scenes, the formalization of the adaptation can be identified through the distance between two scenes. In order to make this, all possible cases of differences between scenes' actuators have to be analyzed.

The first, and the simplest, case here is the situation, when actuators of two scenes are equivalent: all actuators of the original scene are available at the target scene. In this case we have only to compare attributes of our actuators and define, which are different. When all attributes are equivalent, no changes to the original scene and its scenario are required. Otherwise, we have to adopt the scenario according to the idea – all commands, which use attributes of actuators, absent in the target model, have to be dropped from the scenario. After it our original scene will be adopted to the new smart space and can be implemented within it dynamically, without any action by the user.

In order to increase the efficiency of this procedure, we can identify, how many differences are between two scenes: the more of them, the higher the likelihood of having to interfere with the user's manual adaptation of the scene.

Keeping these ideas in mind, we introduce the next definition.

Definition 3: Two scenes *Scene*$_1$ and *Scene*$_2$ are adaptable iff they are compatible and $A_1 \cap A_2 \neq \emptyset$.

Our approach does not impose a more stringent condition similar to sensors (for example, all actuators of the original scene have to be saved in the target one), because an actuator defines only the actions for some specific scene. If one ore more actuators are absent in the target scene, we can remove actions, which depend on them, from the scene scenario. But the nature of the scene will be saved: when a specific condition is fulfilled, the action is provided. It is exactly what can be named the adaptation of the scene to the new smart space.

However, in comparison with the idea of the compatibility of scenes, their adaptability requires a more accurate evaluation. The greater the difference between the sets of actuators of the two scenes, the more actions are needed in order to adapt one of them to another. Therefore, the next definition of the actuator distance can be made.

Definition 4: Actuator distance between two scenes *Scene*$_1$ and *Scene*$_2$ is a value:

$$Dim_A(Scene_1, Scene_2) = \sum_i k_i$$

where

$$k_i = \begin{cases} |F_{a1_i}| - |F_{a2_j}|, if \exists j{:}a2_j = a1_i \\ |F_{a1_i}|, otherwise \end{cases}$$

where $a1_i \in A_1$ and $a2_j \in A_2$.

This definition shows, that the distance between two scenes is defined as the number of attributes of its actuators, which are in the first scene (original), but are absent in the second one (target).

In our case, for example, if in the new space there is no a smart thermometer, we cannot implement previously created scene, because in the new space we cannot check conditions for the activation of the scene behavior. As a consequence, in this situation, we will have to ask a person to adopt the scene manually, identifying the equivalent of the smart thermometer in the new space.

Another situation is, when we have the thermometer in both smart spaces, but in the new one there is a ventilator, which is able only to be turned on/off, without specification of its speed. From the formal point of view, this means, that we have a difference in actuators and the actuator distance between them is 1 (only one attribute – speed – is absent in the new scene in contrast to the original, the others attributes coincide). This fully corresponds to the fact that from the script of our program we have to delete one command, which uses the lost attribute.

As you can see, the proposed concept of distances between scenes allows not only to more effectively determine the possibility of their adaptation to each other, but also to unify the cases when such adaptation can be carried out in automatic mode without user intervention.

VII. Conclusion

In our article we proposed the object-oriented model for the smart objects and corresponding qualitative measures of similarity. Together they allow not only to define the structure of any smart space in unified way, but also provide the opportunity to adopt once created scene of the smart space to new conditions.

Summarizing, we can say that using object-oriented models simplifies the process of adaptation scenarios for the smart spaces. Instead of three stages of this process (discovery, personalization, interaction), only two last can be provided. In addition, the personalization can be organized in semi-automatic way against manual in current conditions.

In contrast to the existing approaches, provided by Fernandez-Montes et al. in [10] and by Strohbach et al. in [11], which are based on fixed programs and notations, our model is platform-independent and is focused on the components, which are important to define the behavior of the smart space.

Thus, the main focus shifts from the technical interoperability to the behavioral interoperability, which consists in unifying the scenarios for different smart spaces at the application and services layers. Furthermore, using the concept of the distance between the scenes, the automation of the adaptation process can be provided.

We consider, that the separation of distances into two classes of sensor distance and actuator distance is really important. The first one shows the ability of the previously created scene to be implemented in the new one, while the second one provides the ability of the evaluation for difficulty of the adaptation realization and allows to evaluate the needed number of transformations between scene scenarios.

The effectiveness the proposed approach is shown by the simple case of adaptation the Smart Home scenario between two different smart spaces with absent objects (sensors, actuators).

The application of the proposed approach is the most useful in spheres of the quickly changing smart spaces, for example in the case of industrial robots and/or smart houses, which have a really high degree of the adaptation need.

The created object-oriented model can be successfully implemented and in other spheres, because corresponds with the Common Information Model (CIM) by Distributed Management Task Force (DMTF) and can be extended to the full Managed Object Format (MOF). It follows from the formal description of the CIM model, which includes instances, properties, relationships, classes and subclasses.

As you can see, the object-oriented model proposed contains all these concepts: a smart object can be interpreted as an instance; its components (sensors and actuators) are the classes, described by the set of attributes and methods (properties), which specify the original smart object and are connected with it (relationships with subclasses); physical implementations of sensors and actuators are the examples of subclasses. This means, that our object-oriented model can be used as a whole formal artifact, representing the domain of IoT.

In the future we expect to extend the created approach by analyzing the interfaces, which can be defined in the case of IoT as a combination of different capabilities of the devices. In order to achieve this goal, the ideas of object-oriented interface representation from [12] can be used.

We also propose to consider the issue of interfaces adaptation by the end users, in native way. For this goal ideas of our object-oriented approach and DSL combination with IoT can be examined.

REFERENCES

[1] F. Bonomi, R.A. Milito, J. Zhu, S. Addepalli, "Fog computing and its role in the internet of things", *in Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, August 2012, pp. 13-16.

[2] B. Negash, T. Westerlund, A.M. Rahmani, P. Liljeberg, H. Tenhunen, "DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices", *Procedia Computer Science*, vol. 109, 2017, pp. 416-423.

[3] B. Negash, T. Westerlund, A.M. Rahmani, P. Liljeberg, H. Tenhunen, "LISA: lightweight internet of things service bus architecture", *in Proceedings of the 6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015)*, June 2015, pp. 436-443.

[4] 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, pp. 693-696.

[5] J. Blom, "Personalization: a taxonomy", *in: CHI'00 extended abstracts on Human factors in computing systems*, 2000, pp. 313-314.

[6] J.C. Augusto, C.D. Nugent, *Designing Smart Homes: The Role of Artificial Intelligence*, Springer, 2006.

[7] 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, pp. 2503-2525.

[8] C. Goumopoulos, A. Kameas, "Smart objects as components of ubicomp applications", *International Journal of Multimedia and Ubiquitous Engineering*, 4 (3), 2009, pp. 1-20.

[9] 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*, 2017, pp. 1-18.

[10] A. Fernandez-Montes, J. Ortega, J. Sanchez-Venzala, L. Gonzalez-Abril, "Software reference architecture for smart environments: Perception", *Computer Standards & Interfaces*, 36 (6), 2014, pp. 928-940.

[11] M. Strohbach, H.-W. Gellersen, G. Kortuem, C. Kray, "Cooperative artefacts: Assessing real world situations with embedded technology", *in: International Conference on Ubiquitous Computing*, 2004, pp. 250-267.

[12] B.A. Johnsson, G. Weibull, "End-User Composition of Graphical User Interfaces for PalCom Systems", *Procedia Computer Science*, vol. 94, 2016, pp. 224-231.

[13] Apple HomeKit, Web: https://www.apple.com/ru/shop/accessories/all-accessories/homekit.

[14] Google Brillo: Android Things, Web: https://developer.android.com/things/index.html.

[15] W3C. Document Object Model (DOM), Web: https://www.w3.org/DOM/.

[16] IEC ISO/IEC JTC 1/SC 41: Internet of things and related technologies, Web: http://www.iec.ch/dyn/www/f?p=103:7:0::::FSP_ORG_ID:20486.

[17] ITU-T Y.2066: Common requirements of the Internet of things, Web: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=12169&lang=ru.

[18] CoAP (RFC7252): Neighbor discovery optimization for IPv6 over low-power wireless personal area networks (6LoWPANs), Web: http://tools.ietf.org/html/rfc6775.

[19] D.J. Cook, M. Youngblood, E.O. Heierman, K. Gopalratnam, S. Rao, A. Litvin, "MavHome: An agent-based smart home", *in: Proceedings of the first IEEE international conference on pervasive computing and communications*, 2003, pp. 521-524.

[20] S. Sebastian, P.P. Ray, "Development of IoT invasive architecture for complying with health of home", *in: Proceedings of I3CS*, 2015, pp. 79-83.