Prottoy Middleware Platform for Smart Object Systems

Fahim Kawsar[1], Kaori Fujinami[2] and Tatsuo Nakajima[1]
[1]*Waseda University, Tokyo, Japan.*
[2]*Tokyo University of Agriculture and Technology, Tokyo, Japan*
{fahim, tatsuo}@dcl.info.waseda.ac.jp, fujinami@cc.tuat.ac.jp

Abstract

This paper presents a middleware platform, Prottoy for the development of smart object systems. A smart object has some specific characteristics e.g., augmentation variation, perceptual feedback provision, push-pull model, etc. In addition a smart object could be stand-alone, co-operative or associated with an external application. Generic pervasive middlewares have no clean support for accommodating all these characteristics of smart objects. Prottoy is designed by carefully observing the characteristics of smart objects and it has taken a core-cloud approach to represent them digitally. A smart object's common functional features are combined together as a generic core and a collection of supplementary service profiles represents the cloud that can be plugged-in atop the core. Application developers are offered unified interfaces to interact bi-directionally (sense-actuate) with the underlying smart objects isolating access issues completely regardless of the smart objects' types and properties. Prottoy's hybrid architecture allows a smart object to be stand-alone, co-operative or a part of an application. The benefit of our approach is twofold. Firstly, the core-cloud artefact framework provides a lucid representation of smart object that accommodates its specific features leading to a generic smart object model. Secondly, Prottoy allows rapid development of smart object systems by providing unified interfaces with high-level abstractions. In this paper, we discuss the background, technical details and the qualitative evaluation of Prottoy.

1. Introduction

One of the consequences of pervasive technologies (e.g., miniaturization of the computer technologies and proliferation of wireless internet, short-range radio connectivity, etc.) is the integration of processors and tiny sensors into everyday objects. This has revolutionized our perception of computing. We are in an era, where we communicate directly with our belongings, e.g., watches, umbrella, clothes, furniture or shoes and they can also intercommunicate. These everyday objects are now designed to provide supplementary services beyond their primary purposes, an initiative that has been denoted as *Smart Object* computing. It has drawn significant attention from the research community; primarily because of its promising potential in various industries e.g., supply chain management, medicine, environment monitoring, entertainment, smart spaces, etc. Another interesting application of smart objects can be seen in its related field of sensor networking. Since, these smart objects are often augmented with multiple sensors, a network of smart objects resembles a sensor network. Consequently, smart object networks could be a practical approach for deploying sensor networks. One additional advantage of such approach is that these objects are facilitating sensing as a supplement to their primary established purposes.

In this paper, we look at the system issues for these smart objects. In particular we discuss two issues 1) a suitable artefact framework for representing smart objects and 2) a middleware that enables the rapid development of smart object systems. Although context-aware computing is often interchangeably used to denote smart objects systems, and several middleware have been proposed in the literature [11, 13, 26, 28], we argue that a complete smart object system cannot be confined with these middlewares' scopes. Three primary reasons are:

1. A smart object's functionalities are highly influenced by the designer, and an object's functionalities are often scenario dependent. The system modeling object's functionality needs to be extremely flexible to accommodate the temporal roles of the object, i.e., an object could be augmented initially with one specific function, and later other functionalities could be added incrementally. Widget model used in existing middlewares are unable to handle such multiple, ad-hoc, and incremental nature of smart objects due to the one-to-one design paradigm.
2. A smart object could be stand-alone, co-operative or part of an application. Although, existing middlewares support application to use multiple smart objects, they have no clean support for stand-alone or co-operative smart objects.
3. Typically in context aware computing, tiny sensors are used to instrument the environment for sensing physical phenomena and back end infrastructures are used to model these sensors' data and to provide proactive services. However, smart objects are often capable of both sensing and actuation. Thus the programming abstractions needed for smart object systems are not equivalent to that of context-aware computing in general.

We will show why a generic context-aware platform is not suitable for smart objects and accordingly present a rapid application development platform, Prottoy to meet the specific requirements of smart objects. Our platform is centered on a self-contained core-cloud artefact framework (artefact wrapper component of Prottoy; see section 5.1) where smart objects common functional features are combined in a core, and its specific supplementary service profiles (so-called smartness) can be plugged atop the core as clouds. This framework supports a pure object oriented design methodology where smart functionalities are accessed via the profile abstraction of an object's digital instance, which is in contrast with context and service abstractions used in existing literature. Prottoy also follows hybrid architecture, where an application layer component (virtual artefact component of Prottoy; see section 5.2) allows infrastructure level support for application developers to interact with the self contained smart objects. Such design allows Prottoy to support the development of stand-alone or co-operative smart objects and also applications that integrate multiple smart objects. The application component of Prottoy isolates all access level complexities and provides generic functions to interact with the smart objects regardless of their types and properties. This makes Prottoy an effective middleware for building and rapid prototyping smart object systems. We will justify our claims by demonstrating a series of smart object systems built using Prottoy and positive feedback from the developers who have utilized Prottoy in their projects.

1.1. Contribution

The contributions of this article are three-fold:

1. By analyzing the characteristics of smart objects, we present a core-cloud artefact framework for representing smart objects. This framework can be used as generic smart object architecture independent of middlewares.
2. We present a middleware Prottoy for smart object systems that enables rapid application development utilizing a cleaner programming abstraction. Smart objects are represented in the application space per se and profile abstraction is used to access its functions (i.e., sensing and actuation).
3. Finally, we present a several smart object systems built atop Prottoy to illustrate the feasibility of our approach and share our development experiences.

In the next section we present an overview of smart objects and their characteristics. Then, we position our research with respect to the related work. Next, we proceed to the design issues and technical details of Prottoy. Then, we show the feasibility of our approach by illustrating three smart object systems. Finally, we report some experiences with smart object systems and conclude the paper.

2. Background: Smart Objects

The Oxford American Dictionary defines the terms *Smart* as "Having intelligence" and *Object* as "A material thing that can be seen and touched". However, in pervasive computing the term *Smart Object* has been used in several contexts. For example: low cost visual tagged objects have been used in augmented reality environment, RFID tagged objects have been used in supply chain management and other enterprise applications. Typically for these objects, intelligence such as perception, reasoning and decision-making is allocated at the infrastructure where only tracking, identification and sharing are done at the object end. Our previous works on Sentient Artefact [19, 14] extend this model by incorporating sensing and perception at the object end while managing reasoning and decision-making at the infrastructure. In more sophisticated cases, intelligence is integrated into the object itself. Examples are Mediacup by Beigl et al. [5] Smart Furniture by Tokuda et al. [30] and Cooperative Artefacts by Strohbach et al. [29]. From a hypothetical point, all these objects can be considered as smart objects if the locality of intelligence is ignored. However, while designing platforms for generic smart objects, it is necessary to understand the scope of the so-called "smartness" of objects. Hence, in the rest of this paper we will consider a smart object as:

"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. Supplementary services typically include sharing object's situational awareness and state of use; supporting proactive and reactive information delivery, actuation and state transition."

All the smart objects mentioned earlier can be rationalized under this annotation. For example: the smartness of a tagged object can be seen in the delivery of its identity information; Mediacup's [5] and Cooperative Artefacts' [29] smartness can be contemplated in sharing situational awareness; AwareMirror's [14] smartness can be observed by its proactive information delegation, etc.

2.1. Characteristics of Smart Objects

Considering the history of smart objects, we have observed that a smart object usually exhibits the following characteristics:

2.1.1. Affordance and Variation in Augmentation: Any physical object - in whatever shape or size - has certain affordances that affect how people use it. These affordances allow people to intuitively come up with new ideas to augment the same object to provide different value added services. A rudimentary reason behind this practice is the scenario specific augmentation. However, it is hard to confine the augmentation scope. Consider, Figure 1 depicting two ideal situations, a) one everyday object capable of playing multiple functional roles and b) multiple physical objects sharing a similar functional role. In Figure 1(a) we have a smart table that can be augmented for two supplementary functions: ambient display and proximity detector (whether some one is in front of it or not). In Figure 1(b) we have a mirror display [14] in a washroom whose functionality can be triggered by any of the three augmented objects, e.g., a toothbrush, a comb or a razor. The suitable augmentation of these objects depends on the underpinned scenario, regardless of the multiple functionalities that can be afforded. These situations signify the fundamental characteristic of a smart object: *"augmentation depends on the affordances as perceived by the designers of the underlying applications."*

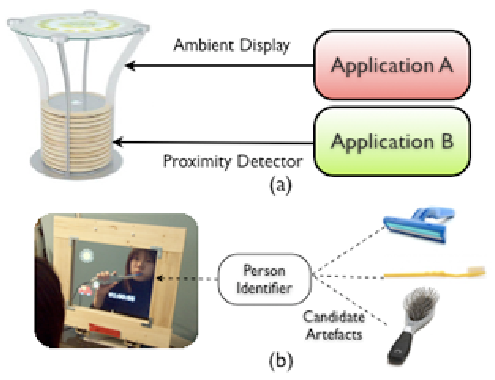

Figure 1: A single object with multiple roles and multiple objects with identical role

2.1.2. Appearance with Perceptual Feedback: Physical objects evolved over the years in physical appearance and acceptances by the end users. A constraint for smart object designers is to keep the original appearance of the objects after augmentation; i.e., augmented services must not be decoupled from the appearance. However, associating computational smartness (in whatever form) to a dumb object also needs to be balanced with the fundamental principles of Human Computer Interaction (HCI), that is to provide perceptual feedback of users' operation and their computational states [6] in a persuasive way. Consider a regular chair, if one of its legs is broken, it is easily perceived by us. What if the chair is augmented with some sensors (so that it can understand someone is sitting on it) and one of the sensors is not working thus affecting its functional output, which cannot be consciously observed. A successful augmentation of the chair provides some feedback to its user by notifying this malfunctioning state to seek attention. Thus, the second characteristic of smart object is: *"keeping its original appearance intact and providing perceptual feedback of its internal state."*

2.1.3. Push-Pull Model: With the rise of *beyond desktop computing*, a significant effort has been made to make the environment aware to provide proactive services [11]. Two approaches have been investigated so far: sensor networking and smart objects. The first has a top down approach; sensor nodes deployed in the physical space are connected to a centralized infrastructure, which deduces the meaning of sensor data for proactive actions.

Here, sensor nodes are used for sensing only and some other artefacts perform the actions. The second, smart objects has a bottom up approach, self-contained objects are responsible not only for collecting the environmental data but also for pushing the environment state via actuation. This distinction is a crucial design factor for intelligent system [24] and corresponding programming abstractions. Smart objects are usually self-contained and can do both: sense the physical phenomenon (pull) and actuate to cause the phenomenon (push). Thus, the third characteristic of smart objects is: *"support for both pushing and pulling the environment phenomenon."*

2.1.4. Object Memory: Adding intelligence to a physical object adds computational memory to the object. This memory content includes but is not limited to static artefact models that describe general properties of an object, dynamic annotations added by the user or an application, and historic information about an object's former states or uses [27]. However, the granularity and locality of this memory varies with the type of augmentation. For example: a tagged objects usually contains identity information in its small non-volatile memory and the underlying infrastructure maintains its state history, where as more sophisticated objects like AwareMirror [14], the states of the object can be maintained locally at the object end. Regardless of the locality i.e., off-board or on-board, smart objects usually associate, advertise, and provide some information stored in its memory. So, the fourth characteristic of smart objects is: *"maintaining a persistent memory either on-board or off-board."*

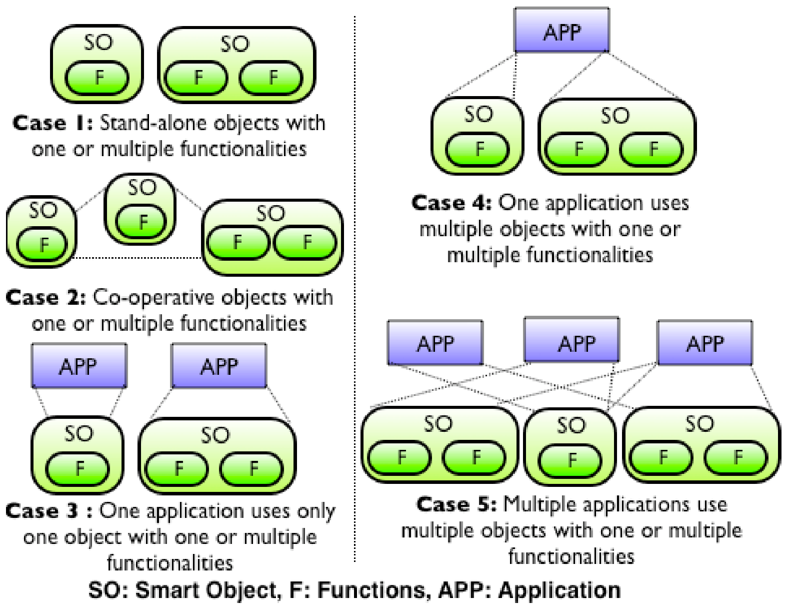
SO
F
Case 1: Stand-alone objects with one or multiple functionalities
Case 2: Co-operative objects with one or multiple functionalities
APP
Case 3 : One application uses only one object with one or multiple functionalities
Case 4: One application uses multiple objects with one or multiple functionalities
Case 5: Multiple applications use multiple objects with one or multiple functionalities
SO: Smart Object, F: Functions, APP: Application

Figure 2: Different use cases for smart object systems

2.2. Smart Object Systems

In general, smart objects operate individually, or are collectively integrated by proactive applications or collaborate with peers to attain a specific purpose. When working collectively a network of smart objects is formed which is often referred to as a smart object system[1]. Henceforth, we observe smart objects systems from three perspectives:

1. Standalone and self contained objects that are independent of any infrastructure and are capable of perception, reasoning and decision making autonomously with appropriate perceptual (auditory or visual) feedback (case 1 in Figure 2). Examples are Mediacup [5], commercial smart objects from Ambient Device [1], etc.
2. Applications integrating multiple smart objects, specifying the interactions between smart objects in order to provide some proactive services (cases 3-5 in Figure 2). This application is executed by another entity that orchestrates the smart objects. Typically a back end infrastructure is utilized by the application for the integration of smart objects. Examples include a smart space with multiple smart objects [19, 17] ambient gaming [25], etc.
3. The third perspective lies in between stand-alone smart objects are also capable of communicating with peers for taking autonomous actions thus creating a co-operative ecology of smart objects (case 2 in Figure 2). Cooperative Artefacts [29] is an example of such an initiative.

In the next section, we look at the related work.

3. Related Work

Prottoy seeks comparison from two aspects: representation and platform for smart objects. Hence, in this section we compare Prottoy from both perspectives.

3.1. Representation of Smart Objects

One of the very first prototypes of smart object was Mediacup [5] where a regular coffee cup was instrumented to provide the state of the cup as context information. Although the Mediacup project and its succeeding Smart-Its [15] provide solid insight into the augmentation of physical artefacts with sensing and processing, they did not provide any generic representation model that can make them usable with any general purpose applications. Tokuda and his group introduced Smart Furniture and u-Textures to build custom furniture [30, 23], however their approach is also closed and tightly coupled with their underlying scenarios. The same is true for other projects in this area where various objects are augmented for providing value added functionalities [1, 29] These objects work fine in a specific scenario, however this assumption of scenario specific objects leads to a less reusable and close development model. The artefact framework presented in this paper takes a generic approach to solve this problem. The core-cloud framework combines the common features of smart objects in a core and allows augmented features to be plugged-in atop the core leading to a generic representation of smart objects in an application independent way.

3.2. Middleware for Smart Objects

Context aware middlewares are typically used to relate with smart objects. Henceforth, in this section we position Prottoy against some context aware middlewares.

3.2.1. Distributed Widget based Middlewares: Distributed component middlewares usually follow a widget-based model [4,11], where underlying objects are represented by distributed

[1] Often the terms "smart object" and "smart object system" are used interchangeably.

widgets that are managed by some widget manager (analogous to GUI widget model) Context Toolkit [11] focuses on the component abstraction by providing the notion of Context Widget and Context Aggregator. Discoverer manages these components and additionally there is a Context Interpreter component that performs the task of context interpretation. Context Toolkit provides one to one mapping for objects' functions to widgets and introduce multiple programming abstractions (widget, service, aggregator) thus making it difficult to manage a smart object system cleanly. Although, for a generic context aware platform Context Toolkit performs very well, it is not suitable for a smart object system, since the abstraction level and smart object requirements are different than what context toolkit was developed for. Thus several components those are specific to a smart object system (e.g., object memory, notification module, profile repository, proxy, storage, etc.) are missing in Context Toolkit. Other distributed middlewares for context aware systems [15, 24] also share these pitfalls. Traditional distributed component technologies like DCOM, CORBA etc. are not suitable for smart object systems due to their unavailability at the range of diverse devices and operating systems.

3.2.2. Infrastructure based Middlewares: Context aware middlewares that have taken a infrastructure approach either in a distributed manner [3,7,18,26,28] or in a centralized manner (black-board architecture) [13] provide fair performance in context acquisition from sensors and in providing interpreted context via standard APIs. This approach has some significant advantages over the distributed widget model since smart objects can share data and services that are independent of underlying hardware. In addition, configuration and evolution of infrastructure-based systems are easier than widget based systems. However, they suffer from single point of failure, scalability and extensibility concerns. Also, collecting information from several sources in one place makes the framework complex and maintenance becomes difficult. Furthermore, this approach cannot support stand alone or co-operative models of smart object systems directly. Prottoy's approach is different from these as it completely distributes the context sources into multiple artefact wrappers (see section 5.1), and provides infrastructure support at individual application spaces through virtual artefacts (see section 5.2). In addition, Prottoy's programming abstractions are cleaner since it presents smart objects as a whole and allows accessing its services (sensing, actuating) and properties via object instance. Furthermore, the discovery process of the distributed smart objects are hidden in Prottoy from the application developers point of view, as there is no centralized networked discovery service like other infra-structured middlewares. In stead, each of the smart objects advertises its services per se which are automatically discovered by the virtual artefact component of Prottoy that runs in the application space.

3.2.2. Middlewares for World Modeling: Several researchers investigated on real world modeling by building context aware middlewares. The Sentient Computing Project [3] utilizes Active Bat location system to provide a platform for indoor applications exploiting a world model. Prottoy's approach is different as it offers the applications to create a context aware environment by constructing an array of smart objects. It means Prottoy specializes the world model creation by allowing developers to construct the model as they want. HP Cool Town [9] encapsulates the world by providing web presence of place, people and thing and allows interaction with web presence of these entities primarily exploiting RF technology. Cool Town supports only web based context aware applications. Easy Living [7] focuses on an architecture that supports the coherent user experience as users interact with variety of devices in a smart environment. Easy Living also utilizes the notion of world model. In contrast to these systems, Prottoy provides a more generic abstraction as developer has the flexibility to construct the model by manipulating virtual artefact.

3.3. Drawbacks of Current Approaches

Earlier in this section, we have looked at several middlewares that provide programming support for various aspects of ubiquitous context-aware computing. However, we argue that these platforms cannot fully accommodate the required features for smart object systems. In the following we present some drawbacks of these systems to justify our claim:

1. **Tightly Coupled Presentation of Smart Object:** Existing infrastructures provide a widget notion to encapsulate the object features [11]. These widgets are not capable of hosting multiple augmented features or do not allow incremental addition of features to a smart object. Adding a new feature to an existing object requires generation of a new widget. This solution is inadequate and impractical because for one physical object, we might end up in multiple widget representation, one for each augmented features.

2. **Inadequate Infrastructure:** Consider, Figure 2 where five different use cases for a smart object are shown. In case 1, smart objects are stand-alone providing a single or multiple built-in functions without any applications. Case 2 shown as example of co-operative smart objects system whereas in cases 3-5 three different modalities of application association are shown. Although the latter cases (3-5) are supported by existing infrastructures [4, 11, 14, 26, 28] by providing a wrapper that is tightly glued with the rest of the infrastructure, but they have no clean support for cases 1 and 2. Smart objects cannot be accommodated natively as stand-alone objects and/or co-operative objects in these infrastructure environments without special care.

3. **Abstruse Programming Abstraction:** Programming abstraction in the existing middlewares [13, 26, 28] is context [10] oriented predominantly. Actuation functions are often presented as action of infrastructure service. Consider, the widget model of Dey et al. [11] Since it follows a one-to-one mapping, if a smart object provides multiple functionalities, for each functions we need a new widget. On the other hand, a service model represents objects that can actuate. Thus for a smart object that can both sense and actuate, we need two different programming abstractions, widget and service. Although, context data can be a service of a smart object, it does not truly reflect what a smart object is or what it is capable of. For example, often the infrastructure service involves the smart object that provides the context. This causes confusion in building applications with smart objects.

4. **Low Reusability:** As mentioned in section 2.1, several features of smart objects (state maintenance, perceptual feedback, etc.) are prevalent. However, due to a missing reusable toolkit for smart object systems that can automatically accommodate these recurring features, developers are required to re-implement these features over and over again.

In the next section, we discuss the design principles adopted in Prottoy to compensate these drawbacks.

4. Design Decisions

Considering the characteristics of smart objects and drawbacks of current approaches, we have adopted a hybrid architecture atop a core-cloud artefact framework in our approach.

4.1. Core-Cloud Artefact Framework for Smart Object

The first characteristic (section 2.1.1) of the smart object essentially signifies that objects' intelligence cannot be confined strictly. On the other hand, all objects do share some common features, which are highlighted by the other characteristics. Accordingly in our artefact framework, the common features of the smart object are built into a core of the artefact model, where the smart features can be added as clouds around the core. The entire model follows a plug-in architecture, whereas the core itself is a generic binary and any cloud (functions) can be plugged into the core. Figure 3, depicts the conceptual core-cloud artefact model. From a designer's point of view, this cloud-based approach gives us the liberty to make the artefact model independent of its perceived affordances, which might be a sensing type (pull) or an actuation type (push). This satisfies the push-pull characteristics of smart objects (section 2.1.3). In addition, these clouds (service profiles) can be plugged into the core anytime thus solving the tightly coupled representation problems of current approaches. This design also solves the absence of a reusable toolkit problem i.e., the core can be shared among multiple applications where as the clouds are scenario-dependent thus allowing the developers to build specialized smart objects. In section 5.1 we present the artefact wrapper component of Prottoy, designed following this artefact model.

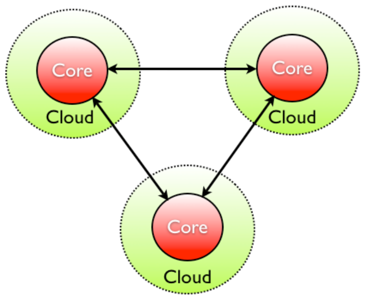

Figure 3: A Conceptual Core-Cloud Artefact Model

4.2. Hybrid Architecture

To accommodate all types of smart object systems as shown in Figure 2, we have adopted this hybrid architecture in our design. The cores of the artefact framework have interfaces to interact with the external worlds and an application layer component called Virtual Artefact (see section 5.2) enables this interaction support to application. In other words, this layer acts as the interface between the external applications and the underlying smart objects. However, the object can work autonomously even if this communication channel at the core is not utilized (stand-alone mode). This hybrid design essentially solves the infrastructure dependency issues of existing systems.

4.3. Smart Object and Profile based Programming Abstraction

The appropriate abstraction for a smart object is the object itself and it can directly be seen as a classic implication of Object Oriented Design. Each of the supplementary functional features of the smart object are represented as independent public procedures (service profiles, see section 5.1) that share some common private data and private procedures (e.g., object memory, notification scheme, etc.) along with the properties of the object (e.g., color, shape, size, owner, vendor, etc.). Application developers, access the instance of the objects that are customized (scenario specific augmentation) for the applications at hand via Virtual Artefact (see section 5.2). This programming abstraction is cleaner than that of existing middlewares used in smart object spectrum and solves the abstruse abstraction problem.

4.4. Application Requirement

Typical distributed middleware attributes are also inherent requirements for a smart object middleware. These requirements include but are not limited to:

1. **Data Dissemination, aggregation and Interpretation:** Middleware platform should have appropriate support for accessing smart objects' data, aggregate them and interpret according to application logic. Our solution incorporates this support at the Virtual Artefact layer (see section 5.2) via lucid APIs. This layer supports both asynchronous (event based subscription) and synchronous (polling) communication modes.
2. **Dynamic Discovery:** Each artefact wrapper (see section 5.1) representing a smart object contains a discovery module at its core that advertises its presence and listens to external requests. A corresponding locator module in the virtual artefact (see section 5.2) layer handles the discovery from the application perspective. However, the actual discovery process is hidden from the application developers through abstract APIs. By distributing the discovery tasks into these two components Prottoy removes the necessity of a dedicated discovery service.
3. **Separation of Concerns and Transparency:** It is obvious that our core-cloud artefact model and hybrid architecture completely separate the applications from the environment and provide applications with transparent accesses through the Virtual Artefact component.

In the next section, we present the architecture and implementation details of Prottoy that follow these design guidelines.

5. Prottoy Middleware Platform

Figure 4 provides a bird eye view of the architecture of Prottoy, which is composed of two components: Artefact Wrapper and Virtual Artefact. The former is the artefact model that encapsulates a smart object whereas the latter is the infrastructure component that allows applications to manipulate the smart objects. For each Artefact Wrapper (smart object), application developers instantiate a Virtual Artefact in the application space to interact with the corresponding smart object. Please note that, although we describe the implementation[2] of these components here, our prototype implementation is not entitled as the only implementation of our architecture. In other words, the Artefact Wrapper design can be thought of an *implementation independent* model. The same applies to the hybrid architecture achieved by the combination of Artefact Wrapper and Virtual Artefact.

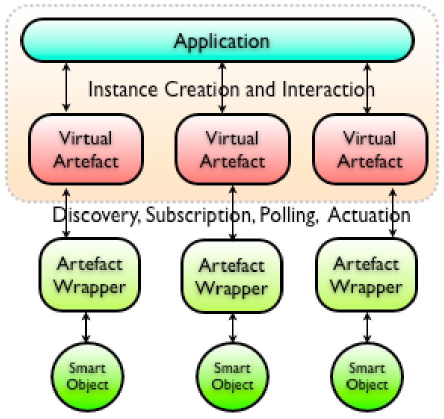

Figure 4: Architecture of Prottoy

[2] Current prototype implementation is done in Java

5.1. Artefact Wrapper

Artefact wrapper provides a layered architecture where basic smart objects' functionalities are combined in a core component as a generic binary. Additional augmented features can be added as plug-ins atop the core. Each augmented feature is called a profile in our approach. These profiles are artefact independent and represent a generic service. This design allows an artefact to be stand-alone artefact and simultaneously participate in an application scenario, thus supporting all use cases of Figure 2. The internal architecture of the artefact wrapper consists of the following (Figure 5):

1. **Core Component:** Typically instrumented artefacts have some common characteristics e.g., capable of communication [5, 29], provides perceptual feedback [6], possesses memory etc. The core component of our artefact framework encapsulates all these functionalities in a generic binary. The *communication module* facilitates communication support and encapsulates the transport layer where as the *discovery module* allows service advertisement. The *notification module* enables the rest of the modules to indicate their status. The *artefact memory* contains property data, profile descriptions, and other temporal data. The *client handler* is the request broker for services and delegates the external requests to specific profiles. Finally, the *profile repository* hosts the array of profiles. The *profile repository* has dynamic class loaders to load the profiles dynamically when requested. The entire core is packaged in a generic binary and runs independently.

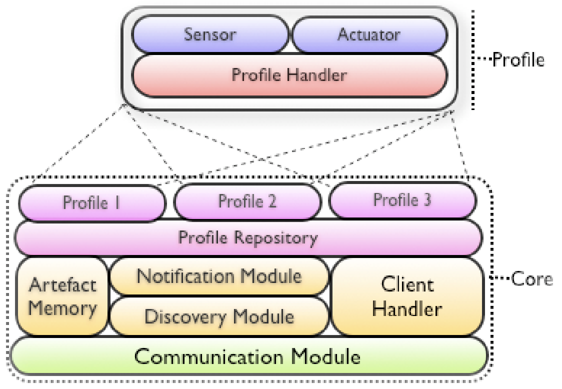

Figure 5: Artefact Wrapper Architecture

2. **Profile:** Each profile represents a specific functionality and implements the underlying logic of the functions, e.g., providing context by analyzing the attached sensors' data or actuating an action by changing the artefacts' state (e.g., increasing the lamp brightness etc.). Each profile is a sensor or an actuator type and has a profile handler, a template to plug device code and context calculation or service actuation logic. The profile handler has an abstraction layer that hides the heterogeneity of the underlying devices. A profile implementation needs to inherit a base `ProfileHandler` class. This class enables the core to load this profile and to further communicate (forwarding application requests etc.) with it. A snippet of a minimal profile implementation code looks like the following:

```
public class ProximityProfile extends ProfileHandler{
/* sensor driver code and context calculation logic */
public void updateContext(){
//do something
setContextData(context);
notify();
}
/* Provide the device driver code service execution code here */
public synchronized Hashtable executeService(Service argument){
//do something}}
```

In the `updateContext()` function (line 3-7), after context calculation, the developer should specify the current context value by setting the context statement using `setContextData()` (line 5) and should invoke `notify()` (line 6), which eventually notifies all the interested clients of this context. In the `executeService()` function (line 9-10) the developer should specify the service actuation logic along with device manipulation code. Interested application can request the service of the profile through appropriate APIs (see section 5.2).

Deployment and Runtime Configuration of Artefact Wrapper: The artefact wrapper is a binary per se with interfaces for external interactions. The profile repository component has a dynamic class loader that loads the plug-ins during artefact instantiation time. Each profile comes with a manifest file along with the profile implementation. This manifest file is used to load the profile. In our current implementation a predefined directory structure is followed, and all the profiles are put in the specific directory. This allows the core to load the profiles using the manifest files.

5.2. Virtual Artefact

Virtual Artefact provides infrastructure support for applications. However, in stead of a dedicated centralized infrastructure as proposed in the existing literature [13,26,28] it runs at the individual application spaces and offers highly abstracted unified interfaces, thus diminishing the drawbacks of a centralized approach. Figure 6 shows the internal architecture of the Virtual Artefact. The *communication module* enables applications to communicate with the artefact wrappers (smart objects). The *locator module* discovers the underlying artefacts wrappers representing the smart objects. The *storage module* enables applications to log objects' data in this storage, maintained as XML database. These logs can be exploited for reasoning purposes. The *proxy module* enables manipulation of the historical data stored in the storage. In addition, application can use this module to simulate a real object utilizing the historical data of the corresponding object. All these module functionalities are complete and can be accessed via highly abstracted APIs. The *Data Processor* module provides data filtering, aggregation and interpretation of profiles outputs (e.g., context data). Application developers implement these components to accommodate application specific processing.

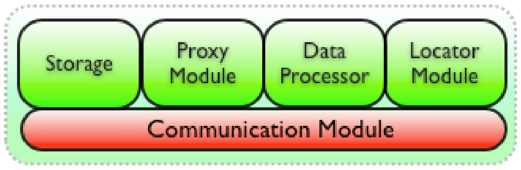

Figure 6:Virtual Artefact Architecture

5.3. Programming Model for Application Development

For building a stand-alone or co-operative smart object system, developers only use the artefact wrapper component. For building applications integrating multiple smart objects, both artefact wrapper and virtual artefact are used. A snippet of a sample application code showing a few APIs usage of Prottoy is shown below.

```
1. VirtualArtefact door =
2.             new VirtualArtefact("Proximity",prop);
3. VirtualArtefact lamp =
4.             new VirtualArtefact("Light",prop);
5. if(door.status){
6.          door.subscribe(this,"doorListener");
7. }
```

```
8. //call back
9. public void doorListener(Context data){
10.    String context = data.getContextData();
11.    /*Interpret according to application logic.
12.    For example for turning on the light we fire */
13.    if(lamp.status){
14.          Service argument = new Service();
15.          argument.setActionName(”switching”);
16.          argument.setCommand(”turn on”);
17.          lamp.execute(argument);
18.   }
```

The sample application uses two smart artefacts, a door augmented with infrared sensors that can sense the proximity of an entity in front of it, and a regular wall lamp that can be turned on/off automatically. For each smart object, an artefact wrapper is generated and deployed. Application creates an instance of a virtual artefact for each artefact wrapper (line 1-4). Application can manipulate the smart objects services (subscribe, poll, execute) via Virtual Artefact APIs (line 5-7, line 13-18). In addition, application can utilize other infrastructure support like data processing, history management, etc. Please note the programming abstraction used here, i.e., the smart object itself in the form of virtual artefact where objects' services are accessed via virtual artefact's unified APIs regardless of the object type. Furthermore, only profile and properties of the objects are used to discover a smart object only. Such high level unification isolates all access level complexities (e.g., discovery, access protocol, marshaling messages, etc.) thus reducing application developers’ burden considerably. Also, the virtual artefact design implements *Reflection* feature of Java rather the event framework to eliminate the strict middleware dependency (e.g., extending a middleware component for event aggregation etc.). In line 6 of the above code the callback handler name was freely defined and implemented. Such simplicity makes application development very simple and rapid using Prottoy.

6. Sample Smart Object Systems

In the introduction section we raised three issues that we addressed in this paper: i) providing a generic artefact framework to represent multi-functional reusable smart objects, ii) providing a middleware that supports different combination of smart object systems (e.g., stand-alone, co-operative and integrating application) and iii) a cleaner programming abstraction for smart objects. We have shown in the earlier sections how our artefact framework provides support for representing multi-functional and reusable smart objects using core-cloud artefact model. We also shown the clean programming abstraction that our approach offers. To address Prottoy's support for different smart object systems in this section we present three systems. The first one is a stand-alone wearable object providing proactive notifications [21]. The second is a co-operative smart object system, where state-of-use information among multiple artefacts is exchanged to form a intelligent living-room [22]. Finally, the third system is a proactive application integrating multiple smart objects where tooth brushing practice is observed for providing persuasive feedback on human lifestyle using a virtual aquarium [25]. We consider these *proof-of-concept* systems qualitatively evaluate the value of Prottoy as suggested by Abwod and Edwards et al. [2,12].

6.1. RoonRoon

RoonRoon (a physical embodiment artefact) is a wearable teddy (as shown in Figure 7) that acts as a user interface for information services [21]. It can monitor user's physical activity state (walking, standing, running and sitting) and can notify personalized information in a contextual manner. RoonRoon's body is augmented with a small wireless sensor node [16], a headset and a host machine. Users can upload their schedule information in its host machine and can provide their notification modality preference. RoonRoon is built using artefact wrapper only, where 3 profiles were used for identifying users activity by analyzing accelerometer data, for allowing user to upload their schedule and for manipulating notification modalities.

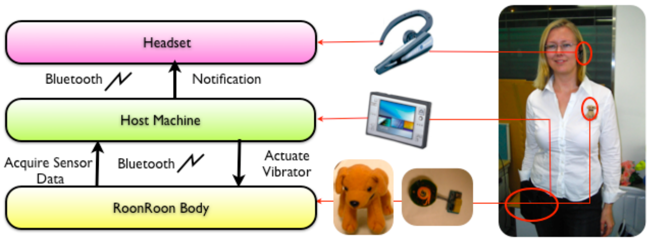

Figure 7: RoonRoon System

6.2. Co-operative Living Room

The second system is for a proactive living room scenario [22]: *"When the door is opened, the lamp is turned on considering room's brightness. If a user picks a phone call, while the TV is on, the TV automatically mutes the volume. The lamp proactively reduces its brightness when the TV is on."* Four smart objects are used in this system (as shown in Figure 8) that are capable of sharing their operational states: Door (open, close), Phone (idle, held), Lamp (on, off, light level), Simulated TV (on, off, volume). All objects (except TV) are augmented with cookie sensor nodes [16] and Gumstix [3] running PC Linux. The lamp is additionally connected to a X10 module. All four objects implemented two profiles each for providing their state-of-use and interacting with peers, where as the lamp and the TV implemented one additional profile to change their states; on/off, brightness, volume level respectively. Only artefact wrapper is used in this application.

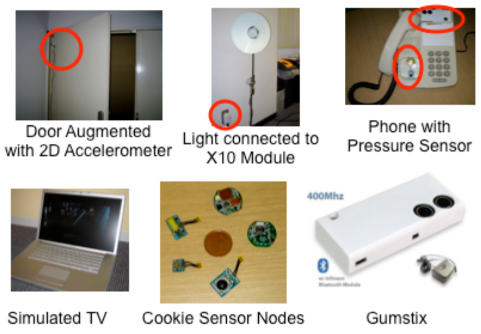
Door Augmented with 2D Accelerometer
Light connected to X10 Module
Phone with Pressure Sensor
Simulated TV
Cookie Sensor Nodes
Gumstix

Figure 8: Co-operative Living Room System

6.3. Virtual Aquarium System

The third application, Virtual Aquarium (Figure 9) has the objective of improving users' dental hygiene by promoting correct tooth brushing practices [25]. The system is set up in the lavatory where it turns a mirror into a simulated aquarium. Fish living in the aquarium are affected by the users' tooth brushing activity. If users

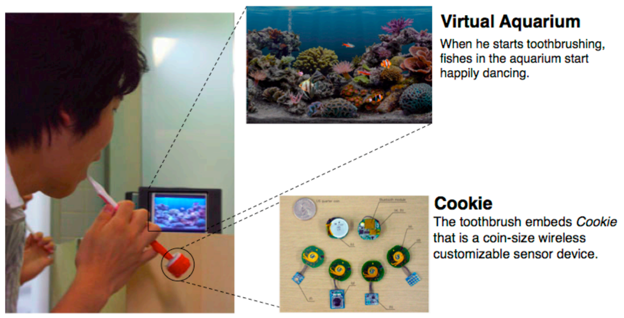
Virtual Aquarium
When he starts toothbrushing, fishes in the aquarium start happily dancing.
Cookie
The toothbrush embeds *Cookie* that is a coin-size wireless customizable sensor device.

Figure 9: Virtual Aquarium System

[3] http://www.gumstix.com

brush their teeth properly, the fish prosper and procreate. If not, they are weakened and may even perish. This application uses a toothbrush augmented with 3D accelerometer sensor that can provide its identity and state of use by implementing one profile. The application monitors tooth-brushing activity by subscribing to artefact wrapper (representing the toothbrush) using virtual artefact and generates appropriate display in the form of an aquarium.

All three systems were successfully built atop Prottoy and deployed over several weeks to understand their usability features that we have reported in [21, 22, 25].

7. Discussion

In this section, first we provide the qualitative evaluation of Prottoy from developers point of view followed by our experience report.

7.1. Evaluation from the Developers Perspective

We have built several smart object systems atop Prottoy. In this paper we reported three systems (section 6) that demonstrate three different use cases for smart object systems. In the following, we mention some premier points identified through these developments and feedbacks from the programmers.

1. **Access Simplicity:** The application code in section 5.3 shows that the virtual artefact removes all access level complexities. The discovery process is completely hidden in Prottoy via lucid APIs. In fact, there is no generic discovery service that runs separately in Prottoy environment. Such high abstractions enable application developers to focus on their application logic rather than spending time on access issues, like discovery, access protocol, message handling etc., which resulted in simple and faster development.
2. **Separation of Concern:** Prottoy's two-layered architecture makes an application well structured and makes an application easy to grow. Developers only define the profiles for distributed artefact wrappers and accumulate those profiles with respective virtual artefacts in application space. An application can be written with some profiles that are absent at the deployment time but could be added at some later phases. Due to the modular structure of Prottoy and plug and play nature of artefact wrapper, such isolation is possible, which allows incremental evolution of an application. This *"separation of concern"* also enabled effective group development; as some developers can focus on the artefact wrappers and others on the virtual artefacts. This fact was observed in multiples times and was also reported by the developers.
3. **Programming Abstraction:** Developers can manipulate the smart objects via virtual artefacts just like other components of their application code (e.g., GUI Class, Math Class, etc.) that are centered on object oriented programming. Virtual artefact behaves completely as an object instance of a class (e.g., having properties, public methods for profile usage, etc.), however in this case the only difference is that it represents a physical and tangible object. This clean abstraction and compatibility with rest of the codes are interesting quality features of Prottoy that are reported by the developers. Also, the APIs used in Prottoy for event manipulation are of free form that allow application code to be structured independent of the middleware, e.g. specific event loop or separate threads are not needed for event manipulation in Prottoy. Such flexibility makes Prottoy very suitable for rapid prototyping.

4. **Reusability:** Since, the core functionalities are shared across multiple smart objects, once the profiles are developed, they can be easily ported to similar objects. Consider the situation depicted in figure 1(b), once the identifier profile is generated for the toothbrush, we can port this profile to the razor or to the comb as long as same sensors are used. This has been one of the major strengths of Prottoy as an application could run in variant scenarios with different objects using the same code.

7.2. Experiences

Over the period of this research, we experienced and realized several interesting issues related to smart objects that we would like to put forward for discussion.

Simplicity and Features: In the earlier prototype of this work we have tried to provide several secondary features, e.g., security, personalization, etc. However, through the development of a series of applications we have realized that these features add little values as most of the applications have their specific needs and defining these features generically at a global scope is very difficult. In fact, for a smart object middleware the primary features i.e., abstracting physical objects, and providing lucid APIs to aggregate events in a simplest way by hiding complexities (discovery, marshaling messages, etc.) are the keys for the developers' satisfaction. For example, Prottoy hides the discovery process completely from the developers. We found this transparency is more important to the developers than providing a versatile separate discovery service at an infrastructure (as we did in our earlier prototype). These issues highlight one significant aspect: *"Secondary features have no value unless the primary features of a middleware are complete and adequate"*.

Performance Metrics: Ubicomp research is experimental in nature and applications are the whole point of ubiquitous computing [8]. This makes it difficult to evaluate a middleware of ubicomp systems. Primarily because the performance metrics typically used to benchmark a distributed middleware are not compelling to measure the quality of a ubicomp middleware. For example, efficiency of a smart object middleware is not constrained by faster throughput or minimum latency, in stead support for proper context identification and triggering of proactive service in a timely fashion are more important metrics for defining efficiency. A smart object is typically battery powered; therefore less energy consumption is a major design goal for smart object middlewares. Generally speaking, a smart object middleware have very little commonalities with traditional distributed system middleware, at least from the benchmarking perspective.

System Robustness is Hidden: A smart object system is often physically distributed and provides proactive services contextually. This characteristic suggests that users attentions' on smart object systems are not coherent. Thus if a particular node (e.g., artefact wrapper or virtual artefact) fails and restarts silently, it is very likely that users will be unaware of that fact. Of course, in situations where users are actively interacting with the system, or if the level of error is very critical, e.g. entire hardware damage, etc., failures will be visible. However, considering the physical nature most of the time the systems' robustness is hidden from the end users.

7.3. Shortcomings of Prottoy

Prottoy is specifically designed for smart object systems. Thus, it is not suitable for a generic context-aware or sensor networking application. Prottoy enables a smart object to have multiple augmented functions and these functions (i.e., service profiles) are derived by the designers of the system. Such profile notion has serious drawback from standardization aspect. Since, we do not have a common vocabulary or ontologies that can be used to define profiles, one pitfall of our approach can be seen in profile based unification. However, by profile abstraction, we are not trying to define the ontology. In stead, we are providing a structure that designers can use to define their own ontology. Of course, defining the conceptual ontology in a standard way is the hardest part not the encoding. We are fully aware of that, and do not claim that Prottoy provides a solution. Our contribution is providing a lucid architecture that can glue such encoding structures with rest of the systems seamlessly.

8. Conclusion

In this paper we presented a middleware platform, Prottoy for smart object systems. By carefully examining the characteristics of smart objects (augmentation variation, perceptual feedback, push-pull model and object memory) and smart objects systems (stand-alone, co-operative and application oriented) we have adopted a core-cloud artefact framework and hybrid architecture for the middleware. The core-cloud model combines the common features of smart objects in a core and allows augmented features to be plugged-in atop that. The hybrid architecture of Prottoy supports development of stand-alone, co-operative and application oriented smart object systems. We have demonstrated the feasibility of our approach through a series of applications and qualitative evaluations. Prottoy is inherently developed for smart object systems, thus the features applicable to a smart object system are the focal points that define Prottoy's strength. The reverse is also true, i.e., Prottoy is not a generic context-aware platform, and thus generic context-aware middlewares can easily be seen as more versatile than Prottoy. The primary contributions of this works are: an artefact model adopting core-cloud design for generic smart objects, the hybrid middleware architecture with cleaner programming abstraction and solid implementation and validity of our design propositions through several real life applications. We consider, our approach provides elegant solutions of the existing problems of smart objects and will be beneficial to the smart object computing community.

References

[1] Ambient devices, url: http://www.ambientdevices.com.
[2] G. D. Abowd. Software engineering issues for ubiquitous computing. In 21st international conference on Software engineering, 1999.
[3] M. Addlesee, R. Curwen, S. Hodges, J. Newman, A. W. P. Steggels, and A. Hooper. Implementing a sentient computing system. In IEEE Computer, 2001.
[4] J. E. Bardram. The java context awareness framework - a service infrastructure and programming framework for context-aware applications. In Pervasive 2005.
[5] 5. M. Beigl, H. W. Gellersen, and A. Schmidt. Media cups: Experience with design and use of computer augmented everyday objects. Computer Networks, special Issue on Pervasive Computing, 35-4, 2001.
[6] V. Bellotti and K. Edwards. Intelligibility and accountability: Human considerations in context-aware systems. Human-Computer Interaction, 16(2-4), 2001.
[7] B. L. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easy living: technologies for intelligent environments. In 2nd International Symposium on Handheld and Ubiquitous Computing, 2000.
[8] Some computer science issues in ubiquitous computing. M. Weiser. Communications of the ACM, 1993.
[9] C. Deborah and P. Debaty. Creating web representations for places. In 2nd International Symposium on Handheld and Ubiquitous Computing, 2000.
[10] A. K. Dey. Understanding and using context. Personal and Ubiquitous Computing Journal, 5(1):4–7, 2001.
[11] A. K. Dey, G. Abwod, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human Computer Interaction, 16(2-4):97–166, 2001.
[12] W. K. Edwards, V. Bellotti, A. K. Dey, and M. W. Newman. Stuck in the middle: The challenges of user-centered design and evaluation of infrastructure. In CHI 2003.

[13] 13. A. Fox, B. Johanson, P. Hanrahan, and T. Winograd. Integrating information appliances into an interactive workspace. In IEEE Computer Graphics and Applications, 2000.
[14] K. Fujinami, F. Kawsar, and T. Nakajima. AwareMirror: A personalized display using a mirror. In Pervasive 2005, 2005.
[15] H. Gellersen, G. Kortuem, A. Schmidt, and M. Beigl. Physical prototyping with smart-its. IEEE Pervasive Computing, 03(3):74–82, 2004.
[16] K. Hanaoka, A. Takagi, and T. Nakajima. A software infrastructure for wearable sensor networks. In IEEE RTCSA, 2006.
[17] A. Helal, W. Mann, H. Elzabadani, J. King, Y. Kaddourah, and E. Jansen. Gator tech smart house: A programmable pervasive space. IEEE Computer, 2005.
[18] J. I. Hong and J. Landay. An infrastructure approach to context-aware computing. Human-Computer Interaction (HCI) Journal, 16(2-3), 2001.
[19] F. Kawsar, K. Fujinami, and T. Nakajima. Augmenting everyday life with sentient artefacts. In 2005 joint conference on Smart objects and ambient intelligence, 2005.
[20] F. Kawsar, K. Fujinami, and T. Nakajima. Prottoy: A middleware for sentient environment. In IFIP Conference on Embedded and Ubiquitous Computing, 2005.
[21] F. Kawsar, K. Fujinami, S. Pirttikangas, and T. Nakajima. RoonRoon: A wearable teddy as social interface for contextual notification. In The International Conference on Next Generation Mobile Applications, Services and Technologies, 2007.
[22] F. Kawsar, M. A. Masum, and T. Nakajima. Applying commonsense to augment user interaction in an intelligent environment. In The 4th IET International Conference on Intelligent Environment (IE08), 2008.
[23] N. Kohtake, R. Ohsawa, M. Iwai, K. Takashio, and H. Tokuda. u-texture: Self-organizable universal panels for creating smart surroundings. In Ubicomp 2005.
[24] G. Kortuem, N. Davies, C. Efstratiou, K. Kinder, M. I. White, R. Hooper, J. Finney, L. Ball, J. Busby, and D. Alford. Sensor networks or smart artifacts? An exploration of organizational issues of an industrial health and safety monitoring system. In UbiComp 2007.
[25] T. Nakajima, V. Lehdonvirta, E. Tokunaga, and H. Kimura. Reflecting human behavior to motivate desirable lifestyle. In DIS 2008, 2008.
[26] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing, pages 74–83, 2002.
[27] M. Schneider. Towards a general object memory. In DIPSO 2007, 2007.
[28] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In 3rd Working IEEE/IFIP Conference on Software Architecture, 2002.
[29] M. Strohbach, H.-W. Gellersen, G. Kortuem, and C. Kray. Cooperative artefacts: Assessing real world situations with embedded technology. In UbiComp 2004.
[30] H. Tokuda, K. Takashio, J. Nakazawa, K. Matsumiya, M. Ito, and M. Saito. Sf2: Smart furniture for creating ubiquitous applications. In International Workshop on Cyberspace Technologies and Societies, 2004.

Authors

Fahim Kawsar is a Ph.D. candidate at the Distributed Computing Lab of Waseda University. He has been working on the design and integration of smart objects since 2004. He received his M. Engg. in Computer Science from Waseda University in 2006. He is a Microsoft Research (Asia) Fellow and a student member of ACM and IEEE.

Kaori Fujinami is an associate professor in the department of computer, information and communication sciences at Tokyo University of Agriculture and Technology. He received a MS in Electrical Engineering and a Ph.D. in Computer Science from Waseda University in 1995 and 2005, respectively. He has been working on activity recognition, smart object systems and human-computer interaction.

Tatsuo Nakajima is a professor of Department of Computer Science in Waseda University. His interests are Operating Systems, Distributed Middleware, Real-time Systems and Ubiquitous Computing.