Environment – Application – Adaptation: a Community Architecture for Ambient Intelligence

Rémi Emonet
Idiap Research Institute
Martigny, Switzerland
remi.emonet@idiap.ch

***Abstract*—This article considers the software problems of reuse, interoperability and evolution in the context of Ambient Intelligence. A novel approach is introduced: the *Environment, Application, Adaptation* (EAA) is streamlined for Ambient Intelligence and is evolved from state of the art methods used in software engineering and architecture. In the proposed approach, *applications* are written by using some abstract functionalities. All *environment* capabilities are exposed as individual services. Bridging the gap between capabilities of the environment and functionalities required by the applications is done by an *adaptation* layer that can be dynamically enriched and controlled by the end user. With an implementation and some examples, the approach is shown to favor development of reusable services and to enable unmodified applications to use originally unknown services.**

***Keywords*-Ambient Intelligence, Architecture, Environment, Application, Adaptation, DCI, SOA, End-User Programming**

I. Introduction

With modern devices and technologies, and with sufficient engineering effort, it is relatively easy to implement smart office and smart home applications. Such applications are usually bound to the considered environment and hard to adapt to a new environment. In the context of Ambient Intelligence, such static application design fails because the user is mobile and the environment evolves continuously. Also, an Ambient Intelligence system is always running and is open: new services (of possibly unknown types) are introduced from time to time. The challenge of software architecture for Ambient Intelligence is to provide a way of maximizing reuse and limiting maintenance. For example, applications should not require any modification or redeployment to handle new service types. Our approach tackles this problem and others.

In this article, we build upon relevant architectural approaches presented in Section II to introduce a new architecture in Section III. We also present, in Section IV, our implementation together with some examples.

II. Approach Foundations: SOA, DCI and others

Our approach can be seen in continuity with previous architectural concepts. In this section, we introduce the architectural concepts that motivate our approach and we provide discussions about other related work.

A. Service Oriented Architectures: SOA

Service Oriented Architectures (SOA) are used in many different contexts ranging from business integration (within and between companies) to Ambient Intelligence. The principle of SOA is to expose software components as “services”. Each service encapsulates a particular functionality and provides access to it through a clearly defined interface.

One important characteristic of SOA is “service discovery”: a service consumer first queries a service repository (or service resolver) to be able to access a matching provider. Most of the service oriented frameworks work with networked services (a notable exception is OSGi, e.g., in [1]). With networked services, one effect of service discovery is to simplify configuration: service consumers only need to know where to find the service repository.

SOA encourages good encapsulation, loose coupling and abstraction. With little effort, it also helps service consumers adapting to runtime events like the absence or disappearance of a particular service. With encapsulation and discovery, SOA makes it possible to replace a service by another equivalent one, providing the same interface.

As in many other domains, a variety of service oriented initiatives have been proposed but no single standard is clearly dominating. Also, even if service based approaches provide a good way of implementing some “dynamic distributed components”, they fail at solving more advanced integration problems. For instance, consider the use case of having an application dynamically (and with no modification) start using services it was not originally designed to use. Such case is typical of an Ambient Intelligence systems where applications and services evolve continuously. Our approach will consider this integration use-case as a common one and not an exception.

The convergence of “Semantic Web” and SOA have been trying to solve the integration problem by letting service designer use their own ontology to describe their services. Ontology alignment methods are then used to make correspondences between services from different providers. Using such correspondence, a service for a given provider can be consumed by a consumer that was designed in ignorance of this particular provider.

In the context of Ambient Intelligence, many projects attempt to integrate different services by building upon both SOA and approaches like the semantic web. Fully automatic service composition and adaptation have been explored, e.g., using multi-agent reasoning as in [2]. Some interesting and well designed approaches are [3] and its evolutions. Also, the soft appliances from [4] envision a systematic decomposition of all existing appliances as independent services. In this vision, end-user programming is used to recreate new innovative appliances from services. One of the main difficulty (and limitation) of end-user programming is to make it both accessible to any end user and powerful enough.

As a conclusion, plain SOA provides a good basis for Ambient Intelligence but it is does not ensure good integration capabilities. We also think that fully automatic approaches are not desired by the end user: these are not optimal and thus can create frustration, and they prevent end users to express their creativity. Classical end-user programming is also too limited to allows at the same time: enabling anyone to customize and innovate with applications, and enabling some users to help in integrating new devices.

B. Data Context Interaction: DCI

Even if it has been studied and practiced since more than 50 years, the domain of software design and engineering is not solved. With time, industrialization methods (waterfall model, UML, etc.) have been created to reduce waste. More recently, “lean” and “agile” methods have taken momentum as they advocate lighter practices and focus on the client.

In our opinion, the most interesting and relevant evolution in recent software architecture and design is the Data Context Interaction (DCI) [5] approach. DCI can be seen as a second attempt to make object orientation (OO) right. The original goal of object oriented programming (and design) was to align the program data model with the user’s mental model. This feature is the key to a good human computer interaction: you cannot hide a bad design behind any interface. This becomes more and more important in Ambient Intelligence where user interaction is augmented.

The main principles of DCI are as follows. The *data* objects have the only responsibility to access data (e.g., from a database or memory). In DCI, any use-case of the software is a piece of code that manipulates some *roles*, which are fully abstract. A use-case uses only a set of roles and never manipulates directly data objects. The concept of *role* together with the *context* are the cornerstone of DCI. A *context* is responsible for doing the mapping of some roles onto some concrete data objects. The context is populated in response to user interaction (e.g., selecting things then clicking on a button) and then the use-case is executed using this context.

As an example, we can consider a banking application with the use case of making a money transfer between two accounts. More precisely, consider the MoneyTransfer use-case: it involves three roles that are the SourceAccount role, the DestinationAccount role and the MoneyAmountProvider role. The MoneyTransfer code will start a transaction, then query the amount to transfer from the MoneyAmountProvider, then call $withdraw$ on the SourceAccount and call $credit$ on the DestinationAccount. The context is created and populated by the application when the user is asked to select a source account (e.g., his CheckingAccount data object) and a destination account (e.g., one of his SavingsAccount) and an amount (e.g., could be just a plain “int” value).

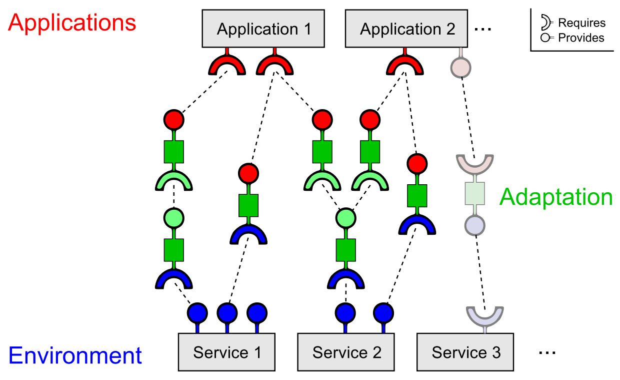

Figure 1. Proposed EAA architecture – *Environment* provides low-level services. *Applications* manipulate only high-level abstract services. *Adaptation* bridges the two and is dynamically extensible and user-controlled. Lighter chain on the right: inversion due to the whiteboard pattern.

C. Other Related Work

A mobile agent is an autonomous programs that can migrate between computers over a network. Even if this is an interesting feature for Ambient Intelligence, it can be seen as orthogonal to the subjects discussed in this article and can complement the proposed approach. An example of using mobile agents as an infrastructure is presented in [6].

The domain of human computer interaction tends to evolve from desktop-like applications to Ambient Intelligence. In this context, an emphasis is put on how to dynamically split and distribute user interfaces based on the available devices. The concept of meta-User Interfaces (meta-UI) has been introduced in [7] and consists in having an interface to control and introspect an Ambient Intelligence environment. A deep and interesting analysis related to our problems is conducted in [7], however, their application is limited to the migration and adaptation of graphical user interfaces between devices.

III. Proposed Approach

In this section, we introduce our Environment, Application, Adaptation (EAA) approach and how it can interact with a community built around it. In the same way as DCI is an attempt to make OO right (see Section II-B), EAA is an attempt to make SOA right.

A. Environment, Application, Adaptation

The Environment, Application, Adaptation (EAA) approach builds on top of Service Oriented Architectures (SOA) and takes similar inspiration as Data, Context, Interaction (DCI). In EAA, most of the elements are services: in some sense, services act as objects (with interfaces) that can be distributed and dynamically discovered. As in SOA, the capabilities of the *environment* are exposed as plain services in EAA. In a parallel with DCI, these environment services are corresponding to the data part from DCI.

Most importantly, EAA has the equivalent of roles in DCI. Any *application* only manipulates some abstract services (roles) that correspond to its exact requirements. The design of the application is done without bothering about what concrete service can or will be used to fulfill the role. With this choice, the environment will never directly provide any service that an application need.

In DCI, the context is responsible for the casting: concrete data objects are recruited to play some roles. In EAA, the *adaptation* layer is responsible for the equivalent, which consists in using services from the environment to create services required by the applications. The adaptation layer is populated through implicit or explicit interaction with the end user (same as in DCI).

In Figure 1 (ignoring the lighter rightmost elements), a set of applications, environment services and adapters are shown. Colors are used to distinguish service types coming from the environment (in blue), the applications (in red) or the adaptation (in green).

B. Using Service Factories for Adapters

To populate the adaptation layer, some adapter factories are used. Each factory is actually a service that exposes which kind of adapters it can create and that creates it on demand. The concept of service factory is taken from [8] and restricted to adapters: we do not consider the case of open factories that can create services without requiring any another service. With our restriction, the number of instantiable adaptation paths becomes finite and it is thus possible to filter and display them to the user (see Section IV-B).

C. Refinement using the Whiteboard pattern

A useful pattern in service oriented design is the "whiteboard" pattern [9]. The goal of this pattern is to simplify the design of clients of a particular service. Let's consider a Text2Speech service that is designed to receive some text sentence and will output it as speech through loud speakers. In a classical approach, any client of the Text2Speech service would first look for the service, then connect to it and then send the message to it. Eventually, the search-and-connect code is here duplicated in all clients.

Using a whiteboard pattern, the situation is reversed and the Text2Speech service is actually doing the search-and-connect. Each client just declares itself as Text2SpeechSource and the Text2Speech will connect to it as soon as it finds it. With the whiteboard pattern, some code is moved from the client to the "server", which limits redundant code writing and makes backward compatible evolutions easier (the server handles the various versions of clients). From a service point of view, now the "server" looks for its clients, which causes an inversion of the provides/requires dependency as shown in Figure 1 (on the right) and in Figure 2 (lower part).

In EAA, the whiteboard pattern is typically used on the view side, i.e., when the application state needs to be brought back to user (through the environment). The above example of voicing the output of an application using a Text2Speech service is a typical example of this.

D. Community Architecture and Sharing

The structure of the proposed EAA makes it a "community architecture" [10] in a double sense. First, the approach encourages the creation of a community around it and provides a structure for it, and second, it is the community itself that is creating the actual, live, evolving architecture.

We distinguish four entry points in EAA for innovation and extension, each requiring different skills. Compared to some end-user programming approach where there is trade-off to make between the expressive power of the programming and the required skills to use it, EAA has multiple values for this trade-off. It would be interesting to investigate how EAA can be combined with an end-user approach targeting more ease of use than power of expression (higher expression power being provided by EAA).

The first two entry points are for a relatively large audience. First, most end users will be able to innovate at the adaptation level by doing a smart and original choice of adapters for a particular application in their environment. Also, any end user can take part in the community by suggesting new ideas for services, applications or adapters. With proper documentations and examples, we can expect a reasonable part of the users (surely less than 10%) to be able to create new adapters by copying an existing one or using a wizard tool (in current implementation, an adapter is just a XML file and thus it is quite easy to define new ones).

More advanced extension points concern the contribution of new applications or new environment services. Both require more advanced computer skills but really different ones. Application developers will probably write their application and maybe a couple of adapters to integrate it into the existing ecosystem: the skills required here are mostly classical application development skills. The contributors of new environment services will probably be people that like hacking with new devices or new signal processing methods (image or audio processing, accelerometers, etc.): their goal would be to innovate by providing innovative input or output medium to transform existing application.

The EAA does not define by itself what kind of services are used by the people. It is the community itself, by creating new environment services, applications and adapters, that decides on what is the actual architecture. We cannot rely on any user to make the best architectural choices. However, if the community is sufficiently large and open, we can expect to find a small proportion of "architects/moderators" as in other open community projects: their role could be for example to avoid proliferation of totally similar concepts and avoid fragmentation of the community.

IV. Implementation and Example

To experiment with the proposed approach, we implemented different test cases. In this section, we provide some implementation details and explain these test cases. More details can be found with the source code that will be provided online, see http://its.heeere.com/ambient2011 .

A. Implementation Details

We implemented the whole presented approach letting aside only the community aspect (e.g., dedicated system for sharing adapters). Our implementation is based on the open-source OMiSCID [11] service-oriented middleware. We created a set of small reusable services and designed a graphical user interface for the user to control the environment and the adapters. The applications are implemented as services that explicitly require some functionalities. Functionalities from the environment are exposed as OMiSCID services.

The developed services will be made available online and include the following services: exporting a display area (on a screen or video project), exporting a mouse pointer, and exporting a "chat" service to allow to open popup messages on a computer. Also, under Linux operating systems, we provide additional features such as a text-to-speech service based on "espeak" and a service to generate synthetic keyboard events on a computer (this one is used for example to control presentations or games).

For the adapters, we designed a generic program that takes an XML description of a family of adapter and starts the corresponding adapter factory (that can start an adapter instance on demand). The XML description contains information about the adapter such as which functionality it takes as "input" and to which one it converts it. The adaptation code, that is usually simple, can be provided within the XML file using ad'hoc languages such as JavaScript or XSLT.

With the assistance of a graphical user interface, the final user can decide what adapters to eventually use. The end of the next subsection is dedicated to the illustration of the simple graphical interface we implemented to help the user managing the environment and the adapters.

B. Detailed Test Case

To showcase our approach, we detail the case of a simple tic-tac-toe game we developed. For now, we consider that

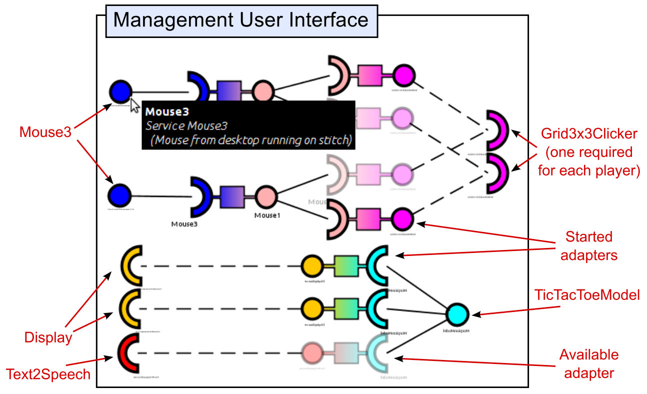
Management User Interface
Mouse3
Service Mouse3
(Mouse from desktop running on stitch)
Mouse3
Mouse1
Grid3x3Clicker (one required for each player)
Started adapters
TicTacToeModel
Display
Text2Speech
Available adapter

Figure 2. User interface to manage adapters (annotated screen capture, best viewed in color). The panel shows all required services, available services and possible adapters obtained from factories using service discovery. A color code is associated to each service type. By clicking on a instantiable adapter (lightened ones) the user can ask the system to create this adapter. Note that due to the whiteboard pattern, the provides/requires relation is reversed for the display side (lower half of the panel) where the environment requires services from the applications.

the environment contains only two computers, and from each one we exported some services: a Display, a Mouse3 (mouse pointer with 3 buttons) and a Text2Speech. Each exported Display service has a unique identifier and follows a whiteboard pattern to connect to any matching DisplaySource it finds. A DisplaySource is expected to send drawing commands to the Display. The game logic is implemented as a service that exposes a TicTacToeModel and that follows a whiteboard pattern for two services: two Grid3x3Clicker with two different unique identifiers.

To bridge the gap between the environment (Display, Mouse3) and the application (Grid3x3Clicker, TicTacToeModel), we introduced a set of simple adapters. The first ones are for input and can be heavily reused in other context: one adapter converts a three button mouse Mouse3 to a single button mouse Mouse1, the second adapter converts a Mouse1 to a Grid3x3Clicker by converting click x, y position to some grid index from 0 to 8. We could have skipped the distinction between Mouse3 and Mouse1 but we kept it as it is useful is some other contexts. On the display side, a specific adapter was written to convert TicTacToeModel to a DisplaySource: the tic-tac-toe state change events are converted to drawing commands such as drawing circles.

From the list of functionalities required by applications and functionalities exposed by the environment, the user interface considers all available adapter factories (also exposed as services, see Section III-B) and proposes possible adaptation paths. Figure 2 illustrates the user interface we designed to allow the user to manipulate adapters. The user sees all possible adaptation paths, and an instantiable adapter is clicked, the system automatically queries the corresponding adapter factory to create the adapter.

By letting the user control the adaptation layer, EAA makes the tic-tac-toe become ambient. The use of properly decoupled services (SOA done right) makes it possible for the user to dynamically select where and how to display the game and how to control it. EAA, with its explicit adaptation layer, makes it also possible to easily create variations of the game that integrates into an Ambient Intelligence vision. To this end, different adapters can be used. A first adapter, which is simple but specific, transforms the game state (TicTacToeModel) to some short textual output to be processed by a Text2Speech service. A reusable adapter, used for input of the game, uses a SpeechRecognizer and converts voice commands such as "play in three" to a Grid3x3Clicker. In addition to the audio modality, computer vision is also used as a possible input: by sticking post-its on a surface, the user can transform it to a Grid3x3Clicker thanks to a dedicated adapter.

C. Other Test Cases

Apart from the tic-tac-toe, we also implemented other environment services, games, applications and adapters. For example we created a MagicSnake game that consists in guiding a snake in a 2D maze to reach a target as fast as possible while avoiding walls. As an experiment, we also modified a game called "Nuncabola" where the player controls a ball rolling in a 3d environment. Both games use a two dimensional analog input: we implemented these input with different combinations of environment services and adapters. Eventually, we control these games using:

- obvious device such as a mouse or a keyboard,
- more exotic devices such a accelerometer-based devices (e.g., smart phone, WiiMote) or WiiFit-like devices,
- computer vision and human tracking (e.g., the player moves in the room to control the ball acceleration, or the player moves his hands, arms, etc.)

Using simple generation of keyboard events, we also implemented a slide presentation controller. We used various methods to skip to the next/previous slide including for example computer vision, e.g., gestures; sound recognition (clapping hands); and voice recognition, e.g., saying "next".

V. Conclusion and Future Work

This article presented the Environment, Application, Adaptation (EAA) architectural approach. With this approach, the environment and the applications are fully independent of each others. This both encourages the design of more generic environment services and eases the deployment of an unmodified application in a new environment: this deployment is possible even if eventually the application ends up using only originally unknown services. The glue between what a particular environment offers and what a particular application requires is done by a dedicated adaptation layer. This layer makes the overall system easier to adapt and open to user control and innovation.

An implementation of this approach was showcased: this implementation is fully operational and allows dynamic runtime extension with new services, applications and adapters.

The main future directions involves the improvement of the user interface (icons for service types, quick filtering, etc), and the setup of the community infrastructure to make it easier for users to use and contribute innovative adapters.

Acknowledgment

The author would like to thank the PRIMA research group and particularly Matthieu Langet for his tight collaboration.

References

[1] C. Escoffier and R. Hall, "Dynamically adaptable applications with iPOJO service components," in *Software Composition*, 2007, pp. 113–128.

[2] M. Vallée, F. Ramparany, and L. Vercouter, "Dynamic service composition in ambient intelligence environments: a multi-agent approach," in *Proceeding of the First European Young Researcher Workshop on Service-Oriented Computing*, Leicester, UK, April 2005.

[3] M. Assad, D. Carmichael, J. Kay, and B. Kummerfeld, "PersonisAD: distributed, active, scrutable model framework for context-aware services," *Pervasive Computing*, pp. 55–72, 2007.

[4] J. Chin, V. Callaghan, and G. Clarke, "Soft-appliances: A vision for user created networked appliances in digital homes," *Journal of Ambient Intelligence and Smart Environments*, pp. 69–75, 2009.

[5] J. O. Coplien and G. Bjørnvig, *Lean Architecture: for Agile Software Development*. Wiley, 2010.

[6] R. Razavi, K. Mechitov, G. Agha, and J. Perrot, "Ambiance: a mobile agent platform for end-user programmable ambient systems," in *Proceeding of the 2007 conference on Advances in Ambient Intelligence*. IOS Press, 2007, pp. 81–106.

[7] J. Coutaz, "Meta-user interfaces for ambient spaces," *Task Models and Diagrams for Users Interface Design*, pp. 1–15, 2007.

[8] R. Emonet and D. Vaufreydaz, "Usable developer-oriented functionality composition language (ufcl): a proposal for semantic description and dynamic composition of services and service factories," in *Intelligent Environments, 2008 IET 4th International Conference on*. IET, 2008, pp. 1–8.

[9] O. Alliance, "Listener Pattern Considered Harmful: The Whiteboard Pattern, 2nd rev." http://www.osgi.org/documents/osgi_technology/whiteboard.pdf, 2004, [Online; accessed 28-July-2011].

[10] F. Moatasim, "Practice of community architecture: A case study of zone of opportunity housing co-operative," Ph.D. dissertation, McGill University, 2005.

[11] R. Emonet, D. Vaufreydaz, P. Reignier, and J. Letessier, "O3miscid: an object oriented opensource middleware for service connection, introspection and discovery," in *1st IEEE International Workshop on Services Integration in Pervasive Environments*, 2006.