Middleware Support for Context-aware Mobile Applications with Adaptive Multimodal User Interfaces

Lincoln David[1], Markus Endler[1], Simone D. J. Barbosa[1], José Viterbo Filho[2]
[1]*Departmento de Informática, PUC-Rio, Brazil*
{lnsilva, endler, simone}@inf.puc-rio.br
[2]*Science and Technology Department*
Universidade Federal Fluminense (UFF), Brazil
jviterbo@id.uff.br

Abstract

Context-awareness has been widely recognized as the key enabler for ubiquitous applications. However, in spite of the recent middleware developments for this field, the design and implementation of such applications is still a major challenge mainly due to the complexity of designing the variant context-specific application behaviors, and structuring the application code so as to enable the seamless switching between the corresponding functions. In order to address these problems we have developed a mobile middleware system on top of Android which eases the development and maintenance of adaptive, context-aware ubiquitous mobile applications. In this paper we present this middleware and show how we used it to implement a prototype of a context-aware Instant Messenger with an adaptive multi-modal user interface. In order to assess the benefit of using our middleware, we compare its implementation with another version of the same messenger application (also with an adaptive and context-aware multi-modal user interface), that we implemented using only the Android framework.

1. Introduction

Context-awareness has been widely recognized as the key enabler for ubiquitous applications. A software is said to support ubiquity if it is able to identify the current state of a mobile user or its device, and automatically adapt its execution behaviour in order to better suit the user's needs. However, in spite of the recent middleware developments for this area, the design and implementation of such applications is still a major challenge. The main reason is the complexity and specificity of (i) using the mobile platform APIs for probing the device's sensors; (ii) deriving application relevant context and/or activity information from this raw sensor data; (iii) implementing the variant, context-specific application functionalities, and (iv) designing and structuring the application code so as to enable the seamless switching between the aforementioned functionalities. These application development challenges are even greater for ubiquitous applications with adaptive (or intelligent), context-aware user interfaces with multiple interaction modes, because of the need to establish timely and non-disruptive transitions between the alternative interaction modes, so that users can keep their focus on their goals and not on the user interface itself.

Most current middleware platforms for context-awareness usually emphasize architectural styles and effective mechanisms for context processing and modeling [1], but do not provide adequate programming level support (e.g. access to context information is simply through APIs). This lack of support causes the resulting context-adaptive application source code to contain several conditionals that are not part of its main logic (i.e. *if-else* control structures spread all over the code), which increases the complexity of development and maintenance. Since these conditionals are *cross-cutting concerns*, a more appropriate programming paradigm, such as Aspect-Oriented Programming [2] or Context-Oriented Programming [3] is required.

In order to address these problems we have developed a mobile middleware system on top of Android which eases the development and maintenance of adaptive, context-aware mobile applications. Our middleware not only supports the composition, reuse, dynamic deployment and interaction between context provisioning modules (i.e. Context Providers), but also provides a programming-level abstraction that eases the coding of context-aware mobile applications in Java. This abstraction, named Context Situation, has been integrated with ContextJ* [3], a Java library for Context-Oriented Programming. It greatly enhances the structuring of an adaptive application, by structuring the variant fragments of application code and bringing developers closer to the definition of usage situations in an application domain. In this paper we present this middleware and show how we used it to implement a prototype of a context-aware Instant Messenger with an adaptive multimodal user interface. In order to assess the benefit of using our middleware, we compare its implementation with another version of the same messenger application (also with an adaptive and context-aware multimodal user interface), implemented using only the bare Android framework.

Thus, the major contributions of our work are:

(i) a design and implementation of a middleware system for configurable and extensible provisioning of context information;
(ii) development of a programming language abstraction and its associated execution mechanism that enables Java programs using ContextJ* to dynamically switch between code variants, which greatly reduces the code complexity of the context-aware mobile applications;
(iii) qualitative evaluation of the advantages of our middleware for application developers through the implementation of a context-aware Instant Messenger with adaptive multimodal user interface;

In the next section, we present a scenario that motivates the use of context-adaptive multimodal user interfaces, as such adaptation capability was the target in our middleware case study. In Section 3, we summarize and compare related work. In Section 4 we then describe the main components of our middleware to support the programming of context-aware applications. Section 5 presents our adaptive mobile application used as case study for the middleware, and in Section 6 we present our qualitative evaluation of its use for the case study. Finally, in Section 8 we draw some conclusions.

2. Context-aware Ubi-Media scenario

Roy works as a car mechanic, and even though more often than not he has his hands busy and dirty with oil, he wants to stay in touch with his friends and girlfriend through instant messaging. Luckily, he has recently found an Instant Messenger for his smartphone which allows him to chat by voice recognition and text-to-speech conversion. So, before starting to work, he activates the voice interface, plugs in the earphones, and puts the smartphone in the chest pocket of his utility uniform. The app also uses a vibratory signal to inform when the other person is typing a new message, so that Roy can wait to hear it before he himself dictates the reply. Roy's girlfriend, Sarah, works at a bike courier service, and also likes to chat with her friends and with Roy during all day. So, she mounts her smartphone on the handlebar of her bike, but uses the speech-capable Messenger in its auto switch configuration, which adds speech mode only while she is pedaling on her bike. So, whenever she makes a stop, the speech mode is temporarily deactivated so that she can read and type in the messages.

Roy and Sarah decided to go to a party that night, and Roy takes his car to pick her up. Now, Roy also sets the Messenger on the auto-switch configuration which in his case means that the app activates speech mode as soon as the smartphone discovers Roy's car radio through its Bluetooth interface. Since Roy is able to continue voice-chatting while driving, before reaching Sara's place, he learns that an old friend is also in town, and decides to pick him up on the way to Sarah, so that all three can go to the party together.

3. Related work

Research in the area of context-aware software has mostly addressed dynamic adaptation from the middleware perspective, exploring architecture styles and the specific mechanisms that best support dynamic adaptation for specific platforms. Along this line, many middleware systems for context-awareness have been developed in the last decade [1]. However, many of them only support deploy-time configuration of context providing components, and not the download, deployment and activation of these middleware elements during execution time, as it is possible in our middleware (cf. Section 4). One exception is the work by Preuveneers and Barbers [4], which present a resource- and context-aware middleware for mobile devices that is component based and self-adaptive. In their system, however, there is no integration with programming-level mechanisms to facilitate the implementation and maintenance of context-aware applications.

There is also much work addressing context-awareness from the programming language perspective, and the most promising approach is probably Context Oriented Programming (COP). It has been explored in many forms, but the main problem with most works is that they are based on specific programming languages - or require specific compilers - that are not available on mobile platforms. For example, ContextL [5] and AmOS [6] are built on top of CommonLisp, and ContextEmerad [7] is based on the concurrent functional language for real-time systems Emerald, while ContextJ [8] is an extension of Java which requires a specific Java compiler.
There are also works that explore the COP paradigm using a variety of scripting languages, such as ContextR, ContextLua, ContextPy [9], or PyContext [2] which are for Ruby, Lua and Python, respectively, and where the COP concepts of layers and dynamic layer activation have been incorporated into these languages. Here, the main problem is that most of these scripting languages either are not available for mobile platforms, or require much run-time resources for these platforms. Appeltauer et al [10] present a quite complete comparison and evaluation of several variants of COP.

Regarding the development of mobile applications with adaptive/intelligent multimodal user interfaces, most research focus mainly on the adaptation of content, and not of the user-system interaction *per se*. Our approach brings together the concepts of universal usability [11] and context-awareness in order to support a wider range of situations, and to do so dynamically, as the user interacts with the system, as opposed to adaptations made between usage sessions, for instance.

4. Middleware components

Our client middleware is structured in two layers: the basic service layer and the programming language layer.

The former consists of Android-based services for asynchronous communication and provisioning of device-local context information, while the latter implements a programming language abstraction and mechanisms that support the dynamic and context-aware switching between application code in Java programs. In the following two subsections, we describe the components of the basic service layer, and in the subsections 4.3 and 4.4 we explain the elements of the programming language layer.

We implemented our middleware on the top of the Android platform because it is freely available, supports Java programming (execution is on the Dalvik VM), defines a powerful Service Oriented Architecture and provides a rich set of APIs and libraries for location-awareness, GUI development and sensor probing.

4.1. Shared Data Manager

The Shared Data Manager, or SDM, is an Android service which implements a publish-subscribe mechanism which is used by application and middleware services alike to exchange data and events locally, i.e. for asynchronous interaction with components deployed on the same device, or remotely, with components and applications executing on remote devices.

SDM can be used for sharing almost any type of data. For publishing a data or event, an application or middleware component only needs to inform the data/event's subject. Optionally, it may inform other properties (attributes), which are used in subscription expressions for filtering the relevant data updates and events. In case of a data publication, it should add a serialized object representing the data itself, e.g. a text fragment, an audio or video object. To receive updates on a specific subject, an application or middleware component must subscribe to the SDM, registering a listener and optionally also informing an expression referring to the data/event properties. Whenever a new publication on this subject happens, the SDM will notify all subscribers of this subject whose expressions match the properties of the published object/event. In order to deliver data/events to subscribers on other devices, SDM interacts with a Pub/Sub broker. This remote asynchronous communication capability of our middleware is further explained in [12].

4.2. Context Management Service

The Context Management Service, or CMS, is an Android service that manages the gathering, processing and distribution of any type of context data. Within CMS, each type of context data/event is *de facto* obtained or produced by a specific Context Provider. Each of such Context Provider (CP) is a component that the CMS can deploy and activate/deactivate independently, depending on whether there is some application component interested in the corresponding context type. CMS also supports the discovery and dynamic download of new Context Providers from a remote Repository of CPs.

CMS uses SDM to deliver the requested context data object to subscribers, regardless if they are local or remote. Context subscribers may be applications or other Context Providers, such as those responsible for transforming or aggregating lower-level context data and producing high-level context information. CMS also provides class ContextConsumer, aimed at hiding from the subscriber/application the code necessary to interact with SDM and CMS, and thus offering a much simpler interface, referring only to the specific context information needed.

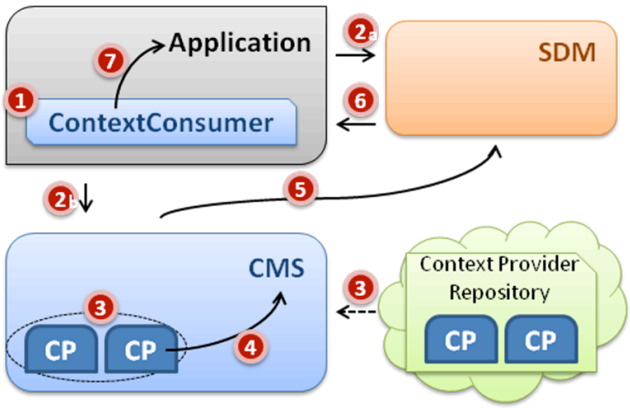

Figure 1: CMS interactions upon new context subscription.

Figure 1 illustrates the basic interactions between an application, the CMS and the SDM: A ContextConsumer of an application issues a subscription to SDM for some specific type of context information, e.g. Battery level, and notifies CMS (steps 1+2). Alternatively, the ContextConsumer may also make a synchronous request to obtain the current state of a specific context type (step 2b). In either case, CMS searches for any locally deployed Context Provider that can produce the requested context information. If none can be found locally, CMS searches, downloads the corresponding CP from the remote repository and activates it (step 3). Once activated, the CP polls or invokes methods at the Android API of the corresponding Resource Manager, and delivers the polled data to CMS as a ContextInformationObject (step 4). CMS adds some data to this object (e.g. the deviceID and timestamp), and publishes it through SDM (step 5). This object is then delivered to all ContextConsumers with matching subscriptions (step 6), which in turn pass it to the applications for specific handling (step 7). Currently, we are setting up a library of Context Providers for CMS that includes following types of context: Geographic location (GPS), Time, Battery level, Type of wireless connection, WiFi RF signal strength, Accelerometer, etc.

4.3. COP and ContextJ

With the use of a context provisioning middleware service (e.g. CMS) the application source code becomes much less mingled with context-specific code snippets. However, the developer still has to code the adaptation's control logic through many conditionals (i.e. *if-else* control structures), which remain scattered throughout the application's source code. In order to mitigate this problem, Hirschfeld *et al.* have proposed the Context-Oriented Programming (COP) [3] paradigm, which is an extension of Object-Oriented Programming and has some similarities with Aspect-Oriented Programming, in that it also supports modularization of cross-cutting concerns. The basic idea underlying COP is to provide means of dynamic activation and composition of *behavioural variations*, by defining alternative source-code segments that can be switched on and off according to the associated context state. Such a variable behavior can be compared to the polymorphism obtained by class inheritance in Object-Oriented Programming, with the difference that the sub-classes are switched at execution time, according to the corresponding context state.

ContextJ [8] is a Java-based COP programming language which extends Java with only some very few constructs, which makes it very easy to learn for Java programmers. The main new structuring concept in ContextJ is called *layer*, which is a named first-class entity used to group all the source code fragments related to a same context-specific functionality. In ContextJ, the layers must be switched on/off explicitly using the commands *with* and *without*, respectively. Thus, there is still no means of toggling between the layers by an element that is external to the application. In our middleware, we use ContextJ*, which is a pure Java implementation of ContextJ's *layers*. The main disadvantage of ContextJ* is that the resulting source code becomes slightly more complex than if using ContextJ's layer structures. However, the advantage is that ContextJ* programs can be executed on any JVM without the need of the ContextJ compiler. And this was the main reason for us to use it on our Android-based middleware, for which there is no ContextJ compiler.

4.4. Situation Evaluation Engine (SEE)

The Situation Evaluation Engine (SEE) is a Java library that enables the application programmer to define Context Situations (i.e. arbitrary complex conjunctions and disjunctions of expressions on context variables), associate listeners to each situation, and link each context situation with a ContextJ* layer. By doing so, these layers can then be automatically activated/de-activated according to the validity of the corresponding context condition. In order to evaluate each registered context situation, the SEE library works in tandem with CMS, and subscribes to all the context types appearing in any such context situation. Then, whenever a new context event is received from CMS, SEE checks the validity of all the registered Context Situations, and for each one that evaluated as true, it calls the corresponding listener and switches to the corresponding layer. The use of listeners thus makes it possible to implement reactive applications. But even if no listeners are used, the activation of some ContextJ* layers will cause the application to change its behavior according to the current context.

In other words, the SEE works as a mechanism that automatically switches between the alternative implementations of a layered method, without the need to implement ContextJ* *with* clauses. Since every layer is linked with a context situation, the platform guarantees the applications behavioral changes exactly in the way the developer planned, by selecting the correct code segment for each method corresponding to the current context on the execution time.

5. The Adaptive Mobile Communicator

Inspired by the Ubi-Media scenario of Section 2, and in order to show the advantages of our middleware, we used it to prototype a mobile application – the Adaptive Mobile Communicator (AMC), which adapts its user interface depending on the user's movement (Figure 2).

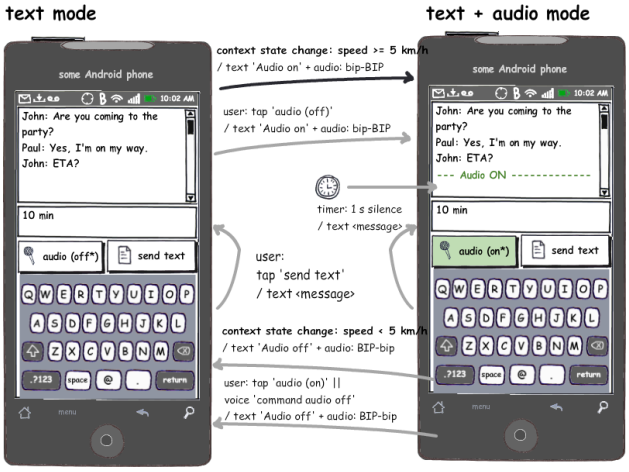

Figure 2: Mockup of the AMC user interface[1].

The AMC is a simple Instant Messenger for the Android platform (version 2.1, or above) and to be executed on any mobile device equipped with a GPS receiver. It uses periodic GPS data readings to infer whether the user is stationary or moving slowly (i.e. less than 5 km/h) or fast. By default, the messaging interface is entirely textual, as found in all traditional messengers. When moving fast, however, an audio interface is switched on, in addition to the textual interface. This

[1] Due to space limitations, we opted to present mockup instead of a screenshot of AMC. Our current prototype is available for download at www.lac.inf.puc-rio.br/~lincoln/amc/amc.html

audio interface uses the Android's text-to-speech functionality for output and the Android 2.1's Voice-Enabled-Keyboard [13] to dictate messages instead of typing them. The rationale of this adaptation is the following: when the user is moving fast, s/he may be engaged in another activity (e.g. driving, riding a bike, or climbing on public transportation), which hinders him/her to focus on the textual interface of the messenger.

In AMC, the context "in fast/slow motion" is computed as follows: (i) the device location is requested to the Android platform; (ii) once the location is acquired by GPS, - or any other mechanism, - the Android informs it to the application; (iii) this location may already carry the speed value (depending on the device), but, if not (iv) the application saves this position on a history queue and uses that queue to calculate the average device speed.

6. Evaluation

To evaluate the advantages of using our middleware, we compared two implementations of the AMC application: one based only on Android, and another using CMS and SEE.

Both implementations share a common core, composed of classes responsible for the user interfaces and for handling communication with a chat server. The *MessageHandler* class, part of the core, is responsible for showing the text messages received from the other participants. Depending on whether the device is moving fast (e.g. more than 5 km/h), this class will also activate the text to speech feature to synthetize the new message. Due to the requirement to adapt to the context, the implementation of this class is slightly different in each version of the AMC, which will be discussed ahead. The core also contains the *RecognitionHandler* class, used to de/activate the speech-recognition feature at the same conditions mentioned above.

In the Android-only version, the *MessageHandler* has the method *setAudioOn(boolean)* which is used by the class *ContextHandler* to de/activate the text to speech functionality. The *ContextHandler* is only present in this implementation version and includes all the Android code to handle device location updates and calculate the device speed. This code is completely platform dependent, and requires specific concepts and knowledge that may not be familiar to most developers. It should be noted that the *ContextHandler* is the element responsible for all the application adaptation, and thus it must have a reference to most classes of the application (including *MessageHandler* and *RecognitionHandler*); which is not a desired class dependency.

In the SEE version, the context-dependent variant functionalities are coded as partial implementations of the *processNewMessage(Message)* in *MessageHandler* class. For the AMC, we defined a layer *OnMovement* and then did a partial implementation of this method that synthesizes the message received. Then we associated this layer with the Context Situation *OnMovementSituation*, enabling that, every time this situation is evaluated as *true,* the corresponding partial implementation will be executed.

Every Situation in the SEE is composed of some SituationRules, but for the AMC only one SituationRule was defined. As the code fragment of AMC shows (Figure 3), this rule is computed based on the context information *this.location.speed*. When the application is launched, the SEE automatically starts the CMS to acquire all context information needed and informs subscribes to updates of the corresponding Context Providers. It should be noted that no extra code is necessary to probe or calculate any context information, since the platform has appropriate services to catch and deliver this information. Thus, with CMS and SEE the development of the adaptive application is much simplified, since the developer does not need to worry about sensor probing and platform-dependent code: he just has to know the name of the required context information.

```
public class Layers {
  public static final Situation
  OnMovementSituation = new Situation() {
   protected SituationRule[]
  defineInitialSituationRules() {
     SituationRule[] rules = new
  SituationRule[1];
  rules[0] = new
    SituationRule("this.location.speed") {
       public boolean getValue(String infoVal) {
        float speed = Float.parseFloat(infoVal);
          if (speed >= 5) return true;
          return false;
       }
     };
     return rules;
   }
 };
 public static final SituationLayer OnMovement =
   new SituationLayer(OnMovementSituation);
}
```

Figure 3: Defining layers.

Table 1 shows some coding complexity metrics used and the values obtained for each version of AMC: Android-only and using our middleware (CMS/SEE). All the values presented in this table concern only the extra code added to the AMC to make it context-aware.

Table 1: Implementation comparison

	Android only	**CMS/SEE**
External classes used	8	always 2
New methods	8	1
External constants	5	1
Concepts to learn	7	2
New lines of code	167	59
Development time	48h	18h
Testing time	12h	2h

The "external classes used" row indicates the number of platform classes needed to be referenced and used by the implementations. Since the SEE version only needs one SituantionRule to define the Situation, the value is always 2. The Android-only version must use the platform specifics to probe sensors, etc. The "new methods" row represents the number of new methods that must be implemented for the platform API interfaces. SEE only dictates that every behavior-variant method must presents a partial implementation version for each adaptation. The number of "external constants" represents the number of constants, like error codes, the application must handle. SEE only returns an error when the information is not available. The "concepts to learn" row represents the number of platform concepts needed to familiarize. In our example, the Android-only version needs: Looper, Listener, Android Services, Android Bundles, GPS activation, GPS configuration; however with SEE we just need Situation and SituationRule. The remaining rows are self-explanatory.

Since AMC requires data from only one sensor (GPS), the values in the left column are relatively low. Of course, this number would increase with the number of sensors required (e.g. accelerometer with GPS), and according to the complexity of the algorithms required for computing high-level context information from the raw sensor data.

While developing the AMC we noticed that the speech recognition function of Android's library is very inaccurate, and that only simple words are correctly recognized. This, of course, makes the application not usable as we hypothesized in our scenario. On the other hand, the text-to-speech function worked pretty well, and all received messages could be understood. For these reasons, we were not able to evaluate the usefulness of adaptive UI by users in a concrete scenario yet.

7. Conclusion

In this paper, we have shown how our middleware, composed of a Shared Data Manager, a Context Management Service and a Situation Evaluation Engine, can support the Context-Oriented Programming paradigm by extending ContextJ* with automatic layer activation. We have also shown that our middleware makes it easier for developers to code their context-aware applications, as opposed to using platform-specific libraries.

A noteworthy limitation of our study is that both versions of the AMC were implemented by the same team that developed the middleware, which could have introduced bias in the results or in the selection of the metrics used in the evaluation. Further studies are needed in which we only briefly instruct programmers on the middleware, to gather additional metrics on the learning curve, and also some subjective data to help to uncover needs or opportunities for refinement.

Additional work is underway to further evaluate our middleware. We hope to increase and refine the complexity measures used to evaluate the cost of programming with the middleware in comparison to platform-specific codes. We are also planning the development of additional adaptive applications to assess the applicability of the middleware, i.e., the range of applications and situations to which it can be applied.

Regarding the AMC, there is still some work to be done in terms of usability, especially regarding system feedback and user control. We are currently conducting a qualitative study to inform the development of a future version of AMC. We hope this study will provide insights and new requirements on the middleware itself, to make it even easier to code usable context-aware applications.

8. References

[1] M. Miraoui, C. Tadj, C. ben Amar, Architectural Survey of Context-aware Systems in Parvasive Computing Environment, Ubiquitous Computing and Comm. Journal, vol. 3, no. 3, 2008.

[2] M. von Lo ̈wis, M. Denker, and O. Nierstrasz. Context-oriented programming: Beyond layers. In Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pages 143–156. ACM Digital Library, 2007.

[3] R. Hirschfeld, P. Costanza, O. Nierstrasz: Context-oriented Programming, in Journal of Object Technology, vol. 7, no. 3, March-April 2008, pp. 125-151.

[4] D. Preuveneers, Y. Berbers, Towards context- aware and resource-driven self-adaptation for mobile handheld applications. Proceedings of the 2007 ACM symposium on Applied computing, 2007.

[5] P. Costanza. Context-oriented programming in contextl: state of the art. In LISP50: Celebrating the 50th Anniversary of Lisp, pages 1–5, New York, NY, USA, 2008. ACM.

[6] S. González, K. Mens, and A. Cádiz. Context-oriented programming with the ambient object system. JUCS, 14(20):3307–3332, 2008.

[7] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. 2010. Programming language support to context-aware adaptation: a case-study with Erlang. In Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10). ACM, pp.59-68.

[8] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara ContextJ: Context-oriented Programming with Java. In Proceedings of the JSSST Annual Conference 2009, 2D-1, Shimane University, Matsue, Shimane, Japan, Sep. 16, 2009.

[9] Hasso-Plattner-Institute Potsdam, http://www.swa.hpi.uni-potsdam.de/cop/ (last visit: February 2011).

[10] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In COP '09: International Workshop on Context-Oriented Programming, pages 1–6, ACM, 2009.

[11] Vanderheiden, G. Fundamental Principles and Priority Setting for Universal Usability. Conference on Universal Usability, 2000. Pages 32-38.

[12] M. Malcher, J. Aquino, H. Fonseca, L. David, A. Valeriano, J. Viterbo, M. Endler, A Middleware Supporting Adaptive and Location-aware Mobile Collaboration, Mobile Context-aware workshop at Ubicomp 2010, Copenhagen, 2010.

[13] Android Developers, Speech Input, http://developer.android.com/resources/articles/speech-input.html (last visit February 2011).