A Middleware Supporting Adaptive and Context-aware Mobile Applications

Lincoln David, José Viterbo, Marcelo Malcher, Hubert Fonseca, Gustavo Baptista and Markus Endler

Abstract Mobile Applications with context awareness features allow mobile users to communicate and share different sorts of context-based information among themselves, such as the current position of other users, geo-referenced data, device speed and others. Although many of such mobile collaboration applications potentially share a good amount of functionality, most of them are developed from scratch, monolithic and tailored to specific mobile platforms, what limits their applicability. This chapter presents a client middleware architecture which supports dynamic deployment and composition of components for context-awareness, common collaboration services, and presents a context oriented language easing the coding of such applications. We also present some prototype context-aware applications — implemented on the top of our middleware services —, one of which was used to assess the advantages of using our middleware.

1 Introduction

Mobile and context-aware collaboration allows geographically distributed mobile users to communicate and share different sorts of context- and location-based information among themselves, such as geo-referenced annotations or the current positions of other users. Although many of such applications may share some functionalities related to context-/location-awareness, communication and sharing mechanisms, most of them are developed from scratch. As a consequence, they are mono-

Lincoln David, Marcelo Malcher, Hubert Fonseca, Gustavo Baptista and Markus Endler
Laboratory for Advanced Collaboration (LAC), Pontifícia Universidade Católica do Rio de Janeiro, R. Mq. de S. Vicente, 225, Rio de Janeiro/RJ, 22451-900, Brazil, e-mail: {lnsilva, marcelom, hfonseca, gustavolb, endler}@inf.puc-rio.br

José Viterbo
Instituto de Computação, Universidade Federal Fluminense, R. Passo da Pátria, 156/Bloco E/3^o andar, Niterói/RJ, 24210-240, Brazil, e-mail: jviterbo@id.uff.br

lithic and include platform dependent-code, what limits their applicability and prevents their portability.

Such restraints may be overcome by the use of middleware platforms, which allow the development of applications that are less dependent of the specific features of mobile platforms and device resources/sensors. Therefore, the development process becomes less complex due to the adoption of high-level software structuring and management mechanisms, as well as the reuse of common modules. Hence, major considerations driving the design of middleware platforms for mobile collaboration are built-in support for flexible deployment and composition of services, context-awareness, flexible and asynchronous service interaction model, data sharing and event distribution, and provisioning of context- and location-aware services.

Nevertheless, while current middleware platforms for context-awareness usually emphasize architectural styles and effective mechanisms for context processing and modeling [1], most do not provide the 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, increasing the complexity of development and maintenance.

In order to address these problems, we have developed — on top of Android — a mobile client middleware that eases the development and maintenance of adaptive, context-aware mobile applications. Our middleware not only supports the composition, dynamic deployment and interaction between application components and context provisioning modules (i.e. small shareable internal components called Context Providers), but also provides a programming-level abstraction that eases the coding of context-aware mobile applications in Java.

This chapter is organized as follows. In the next section we summarize the related work on middleware for mobile context-aware and adaptive applications. In the Section 3, we give an overview of the proposed architecture. Then, in Section 4, we describe in more detail the main middleware level components that we have implemented. In Section 5, we explain the programming-level support given by our system. In Section 6, we give an overview of some of the prototypes that we developed with our middleware. In Section 7, we present a qualitative evaluation of the advantages of using our middleware. Finally, in the last section, we present our concluding remarks.

2 Related Work

From the middleware perspective, research on context-aware software has mostly addressed dynamic adaptation, exploring the architecture styles and the mechanisms that best support dynamic adaptation for specific platforms. Along this line, several 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 [2], which presents 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.

ContextPhone [3] also supports the development of context-aware applications for smart phones. It is composed of interconnected modules which provide a set of open-source libraries and components to be executed on mobile phones. The main modules are the sensor module, which acquires raw context data from different sources, like positioning information from a GPS sensor, and the communication module, which implements connectivity and communication with remote services through different protocols, such as GPRS, Bluetooth, SMS and MMS. However, this platform only provides very simple forms of context-awareness and lacks support for dynamic adaptation of the middleware and applications.

From the programming language perspective, 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 [4] and AmOS [5] are built on top of CommonLisp, and ContextEmerad [6] is based on the concurrent functional language for real-time systems Emerald, while ContextJ [7] 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 or ContextPy [8], in which the COP concepts of layers and dynamic layer activation have been incorporated into Ruby, Lua and Python languages, respectively. 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 [9] present a quite complete comparison and evaluation of several variants of COP.

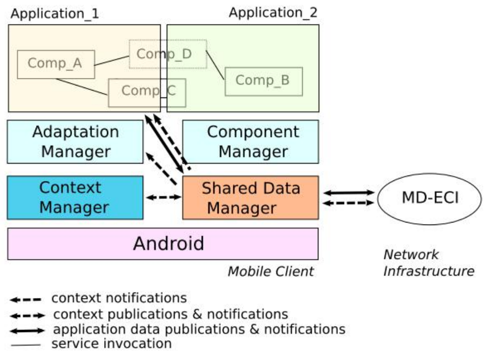


Fig. 1 Client Architecture and MD-ECI

3 Proposed Client Architecture

Our client architecture is composed of a middleware layer and a programming layer, as shown in Figure 1. In our approach, a mobile client application is built out of dynamically deployable and composable components (represented as Comp_X in Figure 1), each of them implementing an elementary functionality of the application (e.g. instant messaging, a map annotation service, etc.). The composition, execution, dynamic adaptation and interaction of these components are supported by five middleware-level services presented ahead. In addition, each component may be implemented using a sixth element, our Context Oriented Programming support, so that the components' behavior may be context-aware and dynamically adaptive as well.

1. Component Manager: This service is responsible for the discovery, dynamic deployment and binding of the components used by applications. It also supports queries about all components currently deployed at the device, and their current states (e.g. loaded, deployed, active/inactive).
2. Adaptation Manager: This service is responsible for triggering dynamic adaptations in the applications components, whenever required. For this purpose, it listens to notifications of context changes, monitors the current configuration of components and requests basic operations on components through the Component Manager. The Component and Adaptation Services are elements of the Kaluana [11] component-based system (cf. Section 4.1).
3. Shared Data Manager: This service provides a uniform API for asynchronous communication among any components, based on a Publish-Subscribe mechanism. A single parameter at the publication operation determines if matchmaking with subscriptions and notifications will be provided only to local subscribers, or also to remote ones. For the latter, SDM relies on MD-ECI.
4. MD-ECI: This service is a SIP-based Publish-Subscribe system [12] that supports remote distribution of notifications of publications — which may be data objects or events — among mobile devices. The main difference between an event and a data object is that the latter is kept in persistent storage at the MD-ECI broker for access by late-joiners, i.e. subscriptions made after the object's publication.
5. Context Manager: This service supports the discovery, deployment and execution of any number of Context Providers. Each of them will collect, process or distribute context data (e.g. resource states or events, and location or sensor data) from the device's mobile platform.
6. Situation Evaluation Engine (SEE): supports the definition of context situations and its association with ContextJ*[7] programming layers, as means of implementing variant and context-specific, dynamically switchable behavior of software components. These context situations and layers define an appropriate programming language abstraction that supports the dynamic and context-aware switching between application code segments.

We chose Android as the primary mobile platform target for our middleware because it supports Java programming, defines a Service Oriented Architecture and provides a rich set of libraries for sensor probing and control, GUI development and Programming Interfaces to Google services, such as Maps or Calendar. The Android programming model defines four essential types of elements that make up a mobile application: Services, Activities, Broadcast Receivers and Content Providers. Although most of our current middleware implementation depends on Android features, all its constituent services can, in principle, be ported to any other service oriented platform, such as OSGi [13].

4 Implemented Middleware Services

In this section we discuss in more detail the basic services of our middleware.

4.1 Kaluana

Kaluana [11] is the tier of our middleware that implements the Component and Adaptation Managers. It defines a component model on the top of Android's service-oriented framework. In this model, each component defines a set of provided services, a set of used services, and the names of other components it depends on. Also, any Android service or activity is component-based, i.e. has a descriptor which defines the set of services it requires for execution. When an activity or service is started, the Component Manager uses the Android framework to search for the required services. If it finds a locally loaded component that implements this service, it simply activates the component. Otherwise, it may download and deploy a suitable component from a remote component repository.

The Adaptation Manager is responsible for determining if a component should be added to, activated, deactivated or replaced from a device, and for selecting the candidate components for such adaptation. This selection is based on the current system context (or user location) and according to an execution pre-requisite associated with each component. For example, when the device switches from a GRPS to a WiFi connection (of a specific and trusted SSID), a component for WiFi RSSI-based (indoor) and site-specific location service may be deployed and activated at the device. In order to actually perform such a dynamic adaptation, the Adaptation Manager issues basic activation and binding requests to the Component Manager, which does the activations and re-bindings and then updates its registry.

4.2 Shared Data Manager

The Shared Data Manager, or SDM, is an Android service that implements a publish-subscribe mechanism which is used by applications and middleware services to exchange data and events both 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 middleware service only needs to inform the data/event's subject. Optionally, it may inform other properties (attributes), which are used in subscription expressions for filtering. In case of a data publication, it should add a serialized object representing the data itself, e.g. a geo-referenced text, video-clip or audio recording. To receive updates on a specific subject, an application or middleware component must subscribe with the SDM, registering a listener and optionally informing also 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 implements also a MD-ECI client.

4.3 MD-ECI

The MD-ECI is a content-based pub/sub system, with a classic centralized architecture. It is composed by a Java API for event publishers and subscribers called Agents, and a Java API, which represents an event broker called Broker. By using the JAIN SIP [15] — a Java API for the SIP protocol — it provides mechanisms for handling client mobility (dynamic changes of their IP address) and periods of client disconnection. More information about the MD-ECI can be obtained from [12].

4.4 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 (CP). Each CP is a component that can be deployed and activated/deactivated independently by the CMS, depending on whether there is an application component interested in the corresponding context type. CMS also supports the discovery and dynamic download of new CP from a remote repository.

CMS uses SDM to deliver the requested context data object to the subscribers, regardless if they are local or remote. Context subscribers may be application specific components or other CPs, such as those responsible for transforming or aggregating lower-level context data and producing higher-level context information. CMS also provides class ContextConsumer, aimed at hiding from the 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.

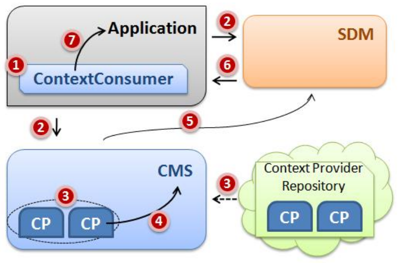


Fig. 2 CMS interactions upon new context subscription

Figure 2 illustrates the basic interactions between an application, the CMS and SDM: A ContextConsumer of an application issues a subscription at SDM for some specific type of context information, e.g. battery level, and notifies CMS (steps 1 and 2); Alternatively, the Context-Consumer may also make a synchronous request to obtain the current state of a specific context type (step 2). In either case, CMS searches for any locally deployed CP 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.

5 Programming Level Support

In this section we introduce the Context Oriented Programming paradigm and explain how we extended it to implement SEE, our programming-level middleware support for adaptive and context-aware application programming.

5.1 COP and ContextJ/ContextJ*

Using a context provisioning middleware component, such as CMS, the mobile application source code already 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) [13] 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 [9] 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.

5.2 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 update 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.

6 Application Prototypes

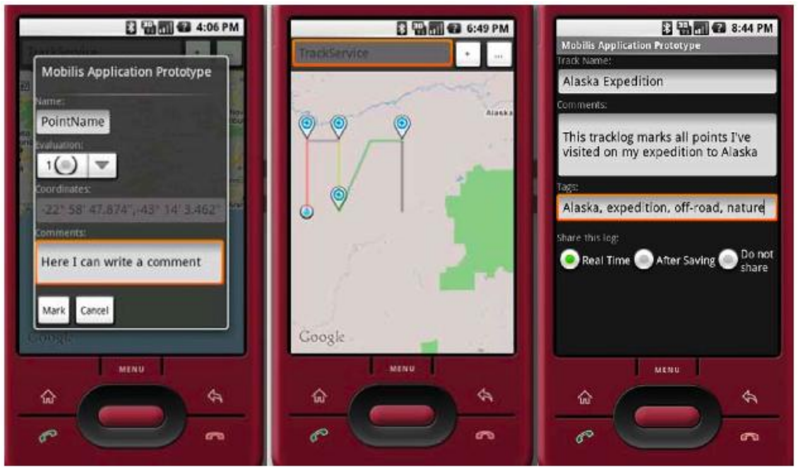
In this section, we give an overview of four prototype applications that we have developed with our middleware.

6.1 TrackService

TrackService is an application used for creating Off-road/Trekking routes/tracks, grading track sections, and sharing this information (i.e. the track log) with other mobile users. The mobile user is able to log his/her route as a sequence of GPS coordinates of his/her mobile device. Figure 3 shows three possible screens of TrackService. While the user is walking/driving along a route, he/she is able to evaluate the route according to the difficulty or danger level, giving a grade from 1 to 10 for each route section (cf. Figure's leftmost screen). As he/she gives grades to the sections of the track, a different color is used to represent each difficulty/danger level, ranging from green (e.g. easy) to yellow, and then to purple (e.g. very difficult). The user can also choose if, he/she wants to share his track with others (cf. rightmost screen), and the options are: in real-time, after saving it on the device, or no sharing.

6.2 ActiveCal

ActiveCal is a location-aware mobile application used for managing meetings. Users running ActiveCal on their smart phones are be able to schedule a meeting, (through Google Calendar) at a specific place and invite some other ActiveCal users. Thereafter, each invited participant is able to track the other participant's position, estimate how long it will take for each one to arrive at the meeting place, and send alerts to the latecomers. The current version of ActiveCal uses Google Maps API and Google Calendar API. The first is necessary for obtaining the distance between two geographic points, the route and the estimated time to reach the destination, while the second provides access to the meeting data, such as date and time, list of participants and meeting place. These pieces of information are collected by specific CMS Context Providers, which combine them to generate a higher-level context information of the sort “Your are late to the meeting at PlaceX”. Because of its component based implementation, it is fairy easy to deploy new calendar or route modules for obtaining the meeting data and the position/distance data from other web services than Google Calendar and Maps.


Mobilis Application Prototype
Name: PointName
Evaluation: 1
Coordinates: -22° 58' 47.874",-43° 14' 3.462"
Comments: Here I can write a comment

Mobilis Application Prototype
Track Name: Alaska Expedition
Comments: This tracklog marks all points I've visited on my expedition to Alaska
Tags: Alaska, expedition, off-road, nature
Share this log: [✓ Real Time] After Saving Do not share

Fig. 3 Screenshots of TrackService

6.3 Bus4Blinds

Bus4Blinds is a mobile application that notifies a person — with visual impairment — waiting for a bus of a selected line when it is approaching the place where he/she is waiting. The notification is symmetric, meaning the bus driver is also notified of the presence of the bind passenger at the bus stop. Thus, the application has two client modes, one for the passenger and one for the bus driver. In this prototype we did not implement any special UI for the passenger mode, but of course, it would have special voice- and audio-based functions for entering/modifying the bus line number. Instead, in our prototype implementation, we exercised the on-line sharing of location information. In fact, Bus4Blind uses our middleware services to track the geographic positions of the driver and passenger clients which have matching bus line numbers. These positions are obtained by GPS ContextProviders and shared in real-time among each other using SDM. As soon as the distance between the matching clients is less than 30 meters, an audio notification is generated at each client. Figure 4 shows three moments of the applications usage: when choosing the client mode (left), when entering the bus line number (middle), and a map-based mode showing the positions of passengers and the bus 301 (right)

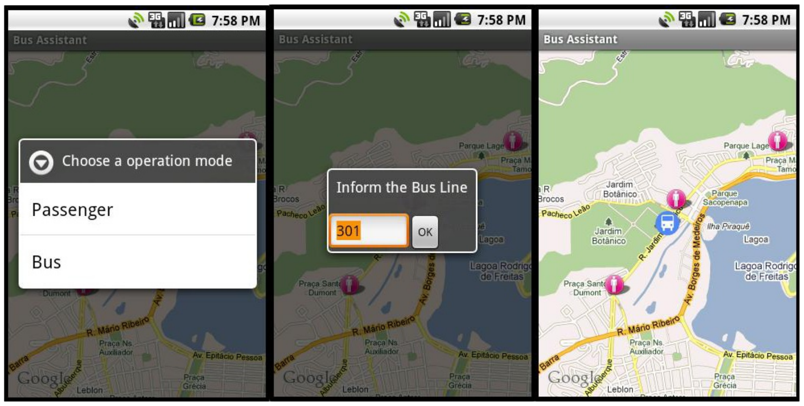
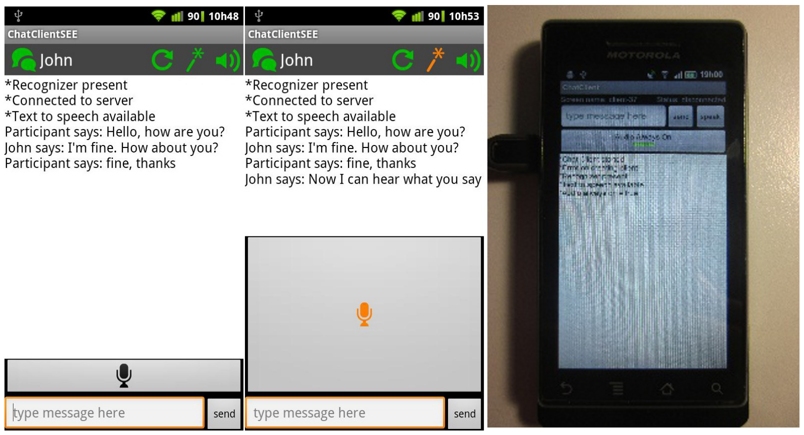


Fig. 4 Screenshots of Bus4Blinds

6.4 Adaptive Mobile Communicator (AMC)

Adaptive Mobile Communicator (AMC) is a simple Instant Messenger for the Android platform (version 2.1, or above) which adapts its user interface depending on the user's movement (Figure 5). 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 audio interface uses the Android's text-to-speech functionality for output and the Android 2.1's Voice-Enabled-Keyboard [14] 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. Figure 5 shows AMC's User Interface with text-based input activated (left), voice-based input activated (middle), and the application executing on a smart phone (right).



*Recognizer present
*Connected to server
*Text to speech available
Participant says: Hello, how are you?
John says: I'm fine. How about you?
Participant says: fine, thanks

*Recognizer present
*Connected to server
*Text to speech available
Participant says: Hello, how are you?
John says: I'm fine. How about you?
Participant says: fine, thanks
John says: Now I can hear what you say

Fig. 5 Screenshots of the AMC

7 Qualitative evaluation of the middleware from the developer's perspective

To assess the advantages of using our middleware, we chose the AMC and compared two implementations of the application: one based only on Android, and another using CMS and SEE. The AMC was choose because we think it explores our middleware features in a more advanced way, since it is a context-aware application that also has a adaptive user interface.

Both application 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*, 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 its 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)* method in *Message-Handler* 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 shown in the code fragment below, 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 about updates of the corresponding Context Providers. It should be noted that no extra code is necessary to probe or calculate any context information, since, as was show in the previous sections, 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 need 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);
}
```

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. 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.

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

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 not very accurate, so more complex sentences in English are not 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 satisfactorily 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.

8 Conclusion

In this chapter, we have shown how our middleware can support the Context-Oriented Programming paradigm by extending ContextJ* with automatic layer activation. This represents a significant improvement in comparison with other middleware approaches discussed in this chapter. We have also shown that our middleware makes it easier for developers to create their context-aware applications, as opposed to using platform-specific libraries.

A noteworthy limitation of our study is that both versions of the application used in our evaluation 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. We are also aware of the scalability problems of our remote context distribution approach and aim at developing a context sharing architecture based on scalable self-managed Publish/Subscribe platform. Additional work is underway to further evaluate our middleware. 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.

The work presented here was done in the scope of the Project ContextNet, which aims at devising and implementing middleware services for efficient and scalable sharing and reasoning of context information from mobile users. Our initial efforts were targeted on developing some basic middleware mechanisms and services, featuring dynamic deployment of components, extensible context-awareness, uniform interface for sharing data, asynchronous communication support (among local and remote components), and providing programming-level abstractions to ease the implementation of applications.

However, there are still many things to be improved, specially with regard to coping with scalability problems. In our view, context information collected from the user's mobile devices will be made available and correlated/combined with data of social networks, so as to derive higher-level information about activities and events of groups of people. Our current research focus on deriving a comprehensive representation of context and situations, developing reasoning approaches and mechanisms for inferring higher-level information and implementing scalable identification and matchmaking mechanisms.

Acknowledgements This work was partially funded by BMBF/CNPq grant 490817/2006-8.

References

1. M. Miraoui, C. Tadj, C. ben Amar, Architectural Survey of Context-aware Systems in Pervasive Computing Environment, Ubiquitous Computing and Comm. Journal, vol. 3, no. 3, 2008.
2. 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.
3. Raento, M., Oulasvirta, A., Petit, R., Toivonen, H. ContextPhone: A prototyping platform for context-aware mobile applications. IEEE Pervasive Computing, 4(2), 51-59, 2005.
4. P. Costanza. Context-oriented programming in contextl: state of the art. In LISP50: Celebrating the 50th Anniversary of Lisp, pages 15, New York, NY, USA, 2008. ACM.
5. S. Gonzlez, K. Mens, and A. Cdiz. Context-oriented programming with the ambient object system. JUCS, 14(20):33073332, 2008.
6. C. Ghezzi, M. Pradella, and G. Salvaneschi. 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, 2010.
7. 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.
8. Hasso-Plattner-Institute Potsdam, http://www.swa.hpi.uni-potsdam.de/cop/ (last visit: February 2011).

9. 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 16, ACM, 2009.
10. OSGi Alliance. 2003. OSGi Service Platform, Release 3. IOS Press, Inc..
11. H. Fonseca, A Component-based middleware for Dynamic Adaptation on the Android Platform, M.Sc. Thesis, Depratment of Informatics, PUC-Rio, 2009.
12. MD-ECI http://www.lac.inf.puc-rio.br/moca/mdeci/mdeci.htm, 2009.
13. R. Hirschfeld, P. Costanza, O. Nierstrasz: Context-oriented Programming, in Journal of Object Technology, vol. 7, no. 3, March-April 2008, pp. 125-151.
14. Android Developers, Speech Input, http://developer.android.com/resources/articles/speech-input.html (last visit February 2011).
15. NIST SIP: http://snad.ncsl.nist.gov/proj/iptel/ (visit July 2011)