An extensible, self contained, layered approach to context acquisition

Dean Kramer
School of Computing and Technology
University of West London
London, UK W5 5RF
dean.kramer@uwl.ac.uk

Anna Kocurova
School of Computing and Technology
University of West London
London, UK W5 5RF
anna.kocurova@uwl.ac.uk

Samia Oussena
School of Computing and Technology
University of West London
London, UK W5 5RF
samia.oussena@uwl.ac.uk

Tony Clark
School of Engineering and Information Sciences
Middlesex University
London, UK NW4 4BT
t.n.clark@mdx.ac.uk

Peter Komisarczuk
School of Computing and Technology
University of West London
London, UK W5 5RF
peter.komisarczuk@uwl.ac.uk

ABSTRACT

Smart phones show increasing capabilities for context-aware applications. The development of such applications involves implementation of mechanisms for context acquisition and context adaptation. To facilitate efficient use of the device's resources and avoid monitoring the same context changes from multiple points, it is necessary that applications share the context acquisition mechanism. In this paper, we intend to develop a generic context acquisition engine which is capable of context capturing, composition and broadcasting. By deploying the engine on a mobile device, context changes are monitored from single point and disseminated to various context aware applications running on the same device. As a proof of concept, the context acquisition engine has been implemented on the Android platform.

General Terms

Design

Keywords

context-awareness, mobile computing

1. INTRODUCTION

Smart phones in recent years have seen high proliferation, allowing more users to stay productive while away from the desktop. This proliferation has seen the increasing amount of mobile applications being developed and becoming available to consumers through centralised application repositories. It has become highly predictable for these devices to have an array of sensors including GPS, accelerometers, digital compass, proximity sensors, sound etc. Using these sensors with other equipment already found in phones, a wide set of contextual information can be acquired.

This contextual information is consumed by context-aware mobile applications. Context aware applications have been described to be *intelligent applications that can monitor the user's context and, in case of changes in this context, consequently adapt their behaviour in order to satisfy the user's current needs or anticipate the user's intentions* [4]. Thus context-awareness is composed of two concepts: context acquisition and context adaptation. Whilst context adaptation can be considered application specific, context acquisition can be application independent by means of monitoring the same contexts by multiple applications. This is expensive and an inefficient use of resources such as battery, CPU and memory. So there is a need for a single customisable context acquisition mechanism which would monitor, manage and disseminate contextual information to context aware applications running on the same mobile device. As the mechanism is used by a wide variety of applications, it should be developed in a generic way. Our goal is to contribute to the design and development of a standalone context acquisition engine capable of monitoring context changes independently from applications and broadcasting context information to various context-aware mobile applications.

In many cases, raw context data is not needed by context aware mobile applications, whereas high-level context information is more applicable. This information can be created by composition of several different context data types. Therefore, the context acquisition engine should support composition of captured context events in the way that the composition of contexts fits for purpose of any context aware mobile applications. We address the problem by implementing a context hierarchy that supports context fusion in a loosely coupled way.

In this paper, design and development of a context acquisition engine is proposed. Its capabilities includes context acquisition, context management, context aggregation and context distribution. The engine has been implemented on the Android platform because the platform is open and provides all the features needed for the engine implementation. This paper is structured as follows: Related work is discussed in Section 2. Architecture of the context engine is described in Section 3. Implementation details of the context engine developed on the Android platform are presented in Section 4. Use of the engine is demonstrated in Section 5. Section 6 discusses the proposed context acquisition solution. Section 7 highlights our further work.

2. RELATED WORK

Context-aware research has gained much interest in recent years. The Context Toolkit has been proposed in [6] by Dey et al., a framework for rapid context-aware application prototyping. This framework is developed around components including *context widgets* for context information retrieval, *interpreters* for abstracting context information, *aggregators* for combining multiple related context information, *services* for executing actions e.g altering activator states, and *discoverers* which maintain a registry of framework existing capabilities.

ContextDroid [9] proposed by Wissen et al. is an expression based context framework for the Android platform. This framework is made up of *context entities* - a collection information from context sensors, *context conditions* - a boolean returned evaluation based on a set of parameters, *evaluators* - interfaces for evaluator methods, and *context expressions* - a method of combining contexts. Particularly with contexts e.g light level its simple to think of different levels (dark, medium, bright) as ranges of the lumens. Using this expression engine would require creating higher level contexts to achieve the same goal as you would need to evaluate each end of the range separately. Also the extensibility of this framework is achieved by accessing the framework source code, compared to our approach adding context components in an application and dynamically using them with the engine at runtime.

Dynamic context composition and modularisation has been targeted in [5]. This approach uses *layers* within the Context-Oriented Programming (COP) language *ContextL* [3] to compose aggregated context behaviour. By defining relationships between the individual contexts, and their adaptation logic, aggregated context behaviour is created. Composition using this approach can be particularly complex and difficult to understand, especially when composing large context aggregations.

In [10], CAMF was presented by Wang and Ahmad incorporating machine learning into a context framework on the Android platform, to evaluate how context-aware systems benefit from machine learning, what context support is provided by Android, and how well suited Android is for the development of context-aware systems. The framework itself is based on the layered architectures described by Miraoui and Baldauf [2].

Gonzalez et al. proposed Subjective-C [7] as context oriented extension to the Objective-C language. This extension allows for easier complex context interrelation expression comparable to [5] though with additions, and adds the ability to express context-dependent method definitions allowing application behaviour adaptation using context. Context-dependent method definitions are handled by finding the method implementation needed for the currently active context, every time the context changes, called *method dispatch.* This language extension similar to other COP languages requires a custom compiler. This approach though targeting COP issues in Objective-C, sheds little light on the use of sensors, and how contexts are interpreted.

Space on mobile devices, though getting larger can be still fairly limited. Therefore, it is desirable to not require a framework to be required for each application, but more share a common context engine across an application family.

3. ARCHITECTURE

In this section, we describe the main conceptual elements of the context acquisition engine and outline a generic context composition mechanism.

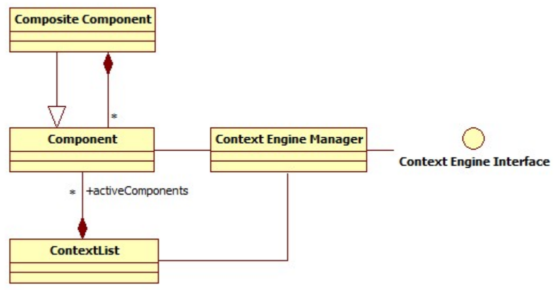


Figure 1: Context Engine Infrastructure

The infrastructure of the context engine is based on the idea of self-contained context components that are structured in a tree hierarchy as depicted in Figure 1. This tree software architecture enables building context hierarchy in which low level context components are loosely composed to form high level composite context. The elements of the infrastructure are described in details below.

3.1 Context Component

Each context component manages a particular context, where context is "*information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves*" [1]. A blueprint of context component is depicted in Figure 2.

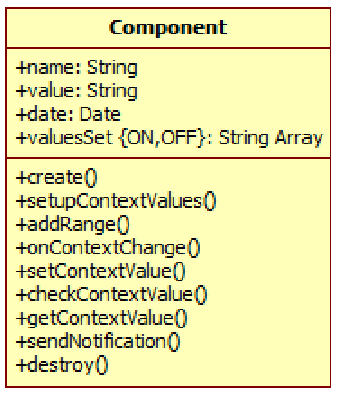


Figure 2: Component blueprint

Component attributes are name, value, date of change and a set of values that it can be set to. Name and value follow

Key-value model which represents the simplest data structure for context information. This model is complemented by date which refers to last context change. The structured context information is sent to mobile applications which are able to identify the context by its name and obtain context value and date of last change. The valid context values of a component are defined by valuesSet. The default set is composed of values *ON* and *OFF*. This set of valid context values can be changed. For instance, Battery context can have defined a set of the valid values: *LOW, OK, FULL.*

3.1.1 Context Acquisition

Each component is self-contained and manages particular context. So it needs to be capable of obtaining raw context data from context sources and dealing with a context change. In order to broadcast meaningful and useful context information, component needs to have an ability to translate raw context data into more meaningful context value. A context change is broadcasted to all listeners if two conditions are fulfilled:

- new context value is an element of predefined set of valid context values;
- new context value differs from previous context value.

For example, if Bluetooth context value set is composed of values *ON* and *OFF* and last context value has been *OFF*, context information is broadcasted if and only if the new context value is *ON*. If context data obtained from Bluetooth adapter is *CONNECTING* then this information is not broadcasted further because it is not element of the set of valid values.

3.1.2 Context Translation

Raw context data can be translated into more meaningful context information. This step is significant in case of numerical raw context data when infinitely many possible context values can be translated into a small finite set of valid values. For example, context data informing about battery level is numerical. In this case, context aware applications are not interested to be informed about every change of battery level, rather they want to obtain more relevant context information such as that battery level is LOW. Therefore, battery context component has to be able to translate numerical values into more meaningful context information. For this purpose, ranges have been created to translate numeric data within a particular range into context value. In respect to battery context, the range for LOW battery context value can be setup as 0% to 10%. Furthermore, by translating raw context data into meaningful context information, it is ensured that number of context values in the values set is finite. This is an important factor for compact context composition and deriving higher level context information as shown below.

3.2 Context Composition

Two or more types of contexts can be loosely aggregated to form high level context. The composed context is managed by a composite component. The constructs of a composite component are shown in Figure 3.

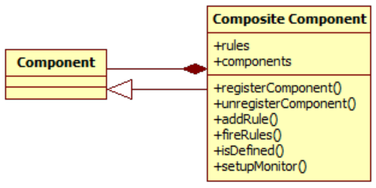


Figure 3: A Composite component

The main role of a composite component is to handle aggregated, high-level context information. Composite components inherit all constructs of context component. Furthermore, each composite component must be aware of contexts it is composed of and rules which drive derivation of high-level context values.

3.2.1 Adaptable composition

Composite components can be constructed, assembled and defined at runtime. Mobile applications may specify and customise high-level context information according to their requirements. This generic approach is driven by rules. The rules are formalized by a function as follows:

Let CC be composite component
Let CV be value set of CC
Let $C_1, C_2, \ldots, C_m$ *be context components*
Let V_i *be value set of component* $C_i, i \in [0, m]$
Let CC be composed of components $C_1, C_2, \ldots, C_n$ *,* $n <= m$
Then rule R can be specified as a function:
f: $V_1 \times V_2 \times \ldots \times V_n \to CV$
Rule $r(v_1, v_2, \ldots, v_n, f(v_1, v_2, \ldots, v_n))$ *where* $v_i \in V_i$

The uniformed rule format enables composition of different types of contexts. Moreover, high-level context information can be defined accurately and precisely. Execution of rules is based on matching of context constants in order to produce the aggregated context value.

3.2.2 Composite Component Behaviour

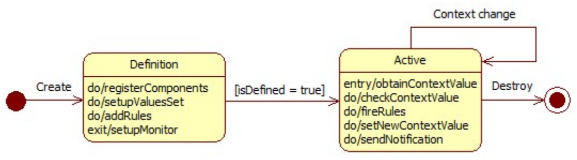


Figure 4: Composite component behaviour

The behaviour of a composite component is illustrated in Figure 4. There are two states of composite component. The initial state is the state of *d*efinition. Definition of composite component involves registering of context components, setting up the relevant context value set and adding rules. Only fully defined composite component can become active. At the moment of leaving the *Definition* state, a listener or monitor of context changes is setup. In the *Active* state, the monitor in the composite component actively listens to context changes broadcasted by registered components.

If a context change occurs, composite component is notified and the new context value is obtained. Rules associated with the context composition are executed in order to obtain corresponding high-level context value. The new high-level context value is set only if differs from previous high-level context value. If a new value is set, a notification about context change is sent to all listeners.

For example, let us have two contexts - *BLUETOOTH* and *WIFI*, both with value sets {*ON,OFF*}. Then a composite component with name *DATASYNC* has value set {*ON, OFF*}. *BLUETOOTH* and *WIFI* contexts are registered in the *DATASYNC* component. Rules are added as follows:

R1({ON,ON}, ON)
R2({OFF,ON}, ON)
R3({ON,OFF}, ON)
R4({OFF,OFF}, OFF)

So if the initial set of context values is <*ON,ON*>, the context value of *DATASYNC* is *ON*. A context change in one context component (*ON -> OFF*) does not change the context value of *DATASYNC* (*ON*) and no notification is sent.

3.3 Context Engine Manager

The role of Context Engine Manager is to manage and distribute context information. The main responsibility of the management part is to control life cycles of context components. Life cycles are defined by the states of components. Components which have entered the active state are stored in a list maintained by the manager. The active components can be further composed and combined. Keeping the list ensures that each component runs only once.

The distribution functionality of the manager ensures that context information is delivered to multiple mobile applications. There are two ways of delivering context information. In first case, context changes are constantly broadcasted to multiple applications. In this *asynchronous* communication, each application is informed basically at ad hoc manner and therefore needs to have implemented a mechanism to distinguish between context information broadcasted by the context engine and other broadcasted data. The second way of delivering context information is direct *synchronous* interaction. In this case, applications can obtain context information on request.

Hence the context acquisition engine performs long-running operations and supplies functionality for other applications to use, the managing element should also operate as a service. To standardise interaction between the context engine service and applications, it is necessary to have a well-defined programming interface with method signatures.

3.4 Context Engine Usage

The overall use of the context acquisition engine is depicted in Figure 5(a). The context engine is deployed on a mobile device as a service with well-defined interface. Context-aware applications can interact with it by using its interface specification. Typically context acquisition logic is same for various context-aware applications. If an application needs to extend the functionality of the components, the context engine can be used as a library within an application as shown in Figure 5(b).

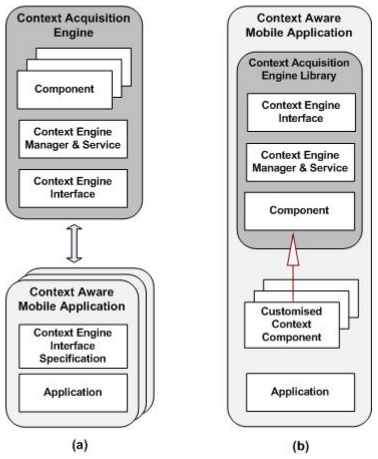

Context Acquisition Engine
Component
Context Engine Manager & Service
Context Engine Interface
Context Aware Mobile Application
Context Engine Interface Specification
Application
(a)
Context Aware Mobile Application
Context Acquisition Engine Library
Context Engine Interface
Context Engine Manager & Service
Component
Customised Context Component
Application
(b)

Figure 5: Usage Scenarios

4. IMPLEMENTATION

As a proof of concept, the context acquisition engine has been implemented on the Android platform. A simple application has been developed to test correct functioning of the engine. This section describes the implementation details.

4.1 Component Types

The context component has been designed in an abstract way to give some conceptual standards to its construction. The methods can be customised and adapted for each context. We have implemented three abstract context component types which inherit the methods from the Component: context monitor, context listener and preference listener.

Mobile device platforms such as Google's Android, Apple's iOS, or Microsoft's Windows Phone are evolving at an amazing rate and increasingly enable higher-level context abstraction. Context information gathered from sensors and networks become more semantic for application developers. Thus there is usually a finite set of predefined values for particular context associated with a particular mobile platform. For instance, Bluetooth context on the Android platform has predefined values *ON*, *CONNECTING* and *OFF*. The context monitor type has been designed for the contexts with a finite set of values. On Android, the context monitor type has implemented a broadcast receiver.

A variety of hardware sensors such as light sensor, accelerometer or GPS in mobile phones act as primary sources of environmental context information. The context listener type has been designed to process raw context data captured by sensors. On Android, this component type implements SensorEventListener interface.

The third type called preference listener has a significant place in the infrastructure. This component has been designed to dynamically configure and alter the execution logic of the context engine at runtime. Preference listener is used to listen to any change in application-specific or global, device-specific user preferences. On Android, preference listener implements OnPreferenceChangeListener.

4.2 Using Android Interface

The context engine is used by third-party applications and needs to be capable of performing long running operations in background. Thus the managing element has been implemented as a service. The service publishes its API through three interfaces declared by using Android Interface Definition Language (AIDL). The interface for context definition enables any mobile application deployed on the same mobile device to specify context hierarchy. This entry point allows defining new composite components and identifying all contexts it is composed of, defining context value sets and adding rules. Composite contexts defined by one application can be used by other applications. The applications can access the list of active contexts and obtain any context information at runtime.

4.3 Context Broadcast Engineering

Messages informing about context changes must be passed at run-time between context components, components and engine, engine and applications. The messages in Android applications which facilitate run-time binding between objects are called intents. A custom, context engine specific intent has been implemented in the Component class. The custom intent is characterised by its action name as shown below on first line in the code snippet. The intent is defined within the sendNotification method in the Component class as follows: Thus a mechanism is needed for listening to con-

```
public static final String CONTEXTINTENT =
        uk.ac.uwl.mdse.contextengine.CONTEXTCHANGED;
public void sendNotification() {
        Intent intent = new Intent();
        intent.setAction(CONTEXTINTENT);
        intent.putExtra(CONTEXNAME, name);
        intent.putExtra(CONTEXDATE,
                Calendar.getInstance().toString());
        intent.putExtra(CONTEXTVALUE, value);
        sendBroadcast(intent);
}
```

text updates in a loosely coupled way. To accomplish this, we have used *Broadcast Receivers*. A broadcast receiver is registered in each component to capture context information broadcasted by other components. The broadcasted intents contain the structured context information of context name, context value and date. This information structuring, broadcasting and receiving form a generic broadcasting mechanism. This mechanism also works for interaction between the engine and applications.

4.4 Dynamic Class Loading

Dynamic class loading (DCL) gives the programmer *the ability to install software components at runtime* [8]. Benefits of using DCL include the capability to lazy load classes, reducing memory usage; the ability to instantiate a component without explicit referencing, allowing more generic code; and finally allow the programmer to add additional context components to the engine without needing to alter or access the engine source code unlike [9]. As shown in figure 5, customised context components can be used by the engine outside of the library. Additionally, as found in Android documentation[1] there is the possibility to extend this to do DCL across multiple applications.

[1]http://developer.android.com/guide/topics/security/security.html

5. EXAMPLE APPLICATION

To demonstrate the use of the context engine within an application, a simple context-aware application has been developed. This application has an interest in obtaining context information about data connectivity. Its context hierarchy

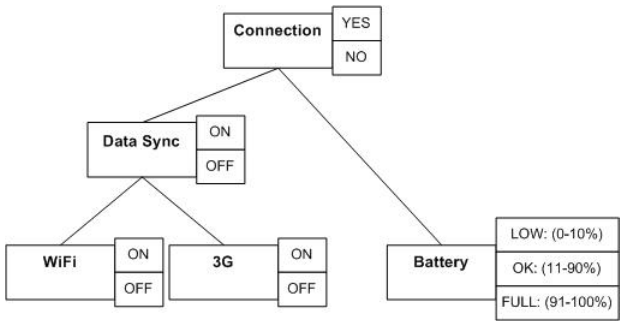


Figure 6: Context Composition

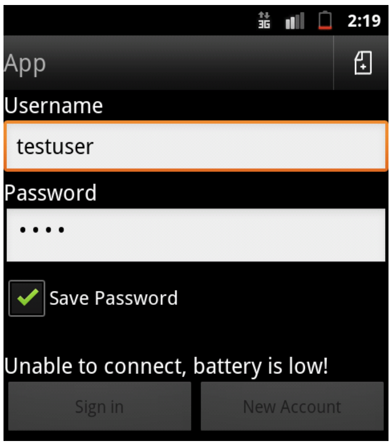
is depicted in Figure 6 and the code snippet for the context hierarchy definition is illustrated as follows:

```
contextService = IContextsDefinition.Stub.asInterface(service);
try {
  contextService.newComposite("DATASYNC");
  contextService.addToComposite("WifiContext","DATASYNC");
  contextService.addToComposite("TelephonyContext", "DATASYNC");
  contextService.addRule("DATASYNC",
        new String[]{"ON","OFF"}, "ON");
  contextService.addRule("DATASYNC",
        new String[]{"OFF","ON"}, "ON");
  contextService.addRule("DATASYNC",
        new String[]{"ON","ON"}, "ON");
  contextService.newComposite("CONNECTION");
  contextService.addToComposite("BatteryContext","CONNECTION");
  contextService.addRange("BatteryContext", "0", "10", "LOW");
  contextService.addRange("BatteryContext", "11", "90", "OK");
  contextService.addRange("BatteryContext", "91", "100", "FULL");
  contextService.addToComposite("DATASYNC", "CONNECTION");
  contextService.addRule("CONNECTION",
        new String[]{"ON","OK"}, "YES");
  contextService.addRule("CONNECTION",
        new String[]{"ON","FULL"}, "YES");
  contextService.startComposite("DATASYNC");
  contextService.startComposite("CONNECTION");
  } catch (RemoteException re) {re.printStackTrace();}
  setupContextMonitor();
...
```

Basically the application specifies the *DATASYNC* component composed of *WIFI* and *3G*. The value sets for all of them are specified as $\{ON, OFF\}$. Hence the application wants to be informed about possible data connectivity only in case that battery level is not *LOW*, it defines the *CONNECTION* component as composition of *DATASYNC* and *BATTERY* contexts. The application defines ranges for the *BATTERY* context and rules for composite components. For instance, if the context value of the *BATTERY* context is equal to *LOW*, the application is notified that the context value of the *CONNECTION* context is equal to *NO*. Thus the application alters its behaviour and in this case, the buttons are disabled as illustrated on the screenshot in Figure 7.

6. DISCUSSION

This paper describes a generic approach to context acquisition for mobile devices. This approach is rule driven enabling application developers to specify high-level context


2:19

App

Username

testuser

Password

••••

Save Password

Unable to connect, battery is low!

Sign in

New Account

Figure 7: Connection - NO

information. Although using rules can enhance complexity management, rules should be executed by a well-designed rule engine capable of handling rule conflicts and circularity. In addition, number of rules can be decreased, for instance, by adding conditions such as ALL values or ANY value must match.

As specified earlier, context changes are propagated up the tree and to the application through the use of broadcasted messages. During the testing and debugging phase, we used the logging system to monitor these broadcasts, and found that in certain conditions the context changes are highly frequent and it results in a considerable amount of messages broadcasted. Though we observed no obvious slow down in system responsiveness, further analysis to make broadcasting more efficient will be required.

Though the use of classloaders has helped separating the engine from each context component and promotes extensibility, there are drawbacks of this approach. When dynamically loading a component, there are assumptions on the class constructor, and any arguments needed, if the constructor is parameterised. Our component constructors require a Android Context[2] object either from the application or activity, for use with the sensors which could limit the application developer.

Other interesting results found were in the area of security. Android uses *permissions* as a way of controlling protected operations. To access sensitive data and access sensors, an application needs the appropriate permission declared in the manifest, which is then used at install time to inform the user of what the application has access to. If the engine is bundled with an application, the permissions needed are added, but if the engine is used externally, then that outside application wont require the permissions as protected operations will be executed indirectly.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented an infrastructure of the context acquisition engine. One of our main goals has been separating context acquisition from context adaptation logic in the way that it can be used by multiple context aware applications running on the same mobile device. We have adopted a simple context aggregation approach which allows dynamic composition of different types of context. We have implemented the context acquisition engine on the Android platform and demonstrated its use by developing a simple mobile application. Our future work will investigate a more precise mechanism for context aggregation based on structured context values. Furthermore, we intend to fully incorporate user preferences as configuration data in order to enforce quality of context aggregation.

[2]http://developer.android.com/reference/android/content/Context.html

8. REFERENCES

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, 1999. Springer-Verlag.

[2] M. Baldauf and S. Dustdar. A survey on context-aware systems. *Interational Journal of Ad Hoc and Ubiquitous Computing*, 2:263–277, 2007.

[3] P. Costanza. Context-oriented programming in contextl: state of the art. In *Celebrating the 50th Anniversary of Lisp*, LISP50, pages 4:1–4:5, New York, NY, USA, 2008. ACM.

[4] L. M. Daniele, E. Silva, L. F. Pires, and M. Sinderen. A soa-based platform-specific framework for context-aware mobile applications. In W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, C. Szyperski, R. Poler, M. Sinderen, and R. Sanchis, editors, *Enterprise Interoperability*, volume 38 of *Lecture Notes in Business Information Processing*, pages 25–37. Springer Berlin Heidelberg, 2009.

[5] B. Desmet, J. Vallejos, and P. Costanza. Layered design approach for context-aware systems. In *in 1st VaMoS 07*, 2007.

[6] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16:97–166, December 2001.

[7] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-c: bringing context to mobile platform programming. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 246–265, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. *SIGPLAN Not.*, 33:36–44, October 1998.

[9] B. van Wissen, N. Palmer, R. Kemp, T. Kielmann, and H. Bal. ContextDroid: an expression-based context framework for Android. In *Proceedings of PhoneSense 2010*, Nov. 2010.

[10] A. I. Wang, B. Wu, and S. K. Bakken. Camf - context-aware machine learning framework for android. In *Proceedings of the International Conference on Software Engineering and Applications (SEA 2010)*, CA, USA, November 2010.