Context-aware Media Player (CaMP): Developing context-aware applications with Separation of Concerns*

Nearchos Paspallis
University of Cyprus
CY-1678, Nicosia, Cyprus
nearchos@cs.ucy.ac.cy

Achilleas Achilleos
University of Cyprus
CY-1678, Nicosia, Cyprus
achilleas@cs.ucy.ac.cy

Konstantinos Kakousis
University of Cyprus
CY-1678, Nicosia, Cyprus
kakousis@cs.ucy.ac.cy

George A. Papadopoulos
University of Cyprus
CY-1678, Nicosia, Cyprus
george@cs.ucy.ac.cy

ABSTRACT

The constant advent of powerful mobile devices has raised the potential of building novel context-aware applications. These applications let the users enjoy a better experience by sensing their context and automating tasks that would otherwise require significant user attention. This paper presents two context-aware applications, built on top of the MUSIC middleware's context management framework. By describing the development steps, we reveal how development is facilitated via Separation of Concerns and how code reuse is enabled in terms of reusable context plug-ins. It is argued that this approach reduces the required development and maintenance effort and thus lowers the associated cost.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping, Reusable software*

Keywords

Context-awareness, Separation of Concerns, Code reuse, Context sensors, Context plug-ins, Mobile applications, OSGi

1. INTRODUCTION

In recent years, the mobile phone proved to be the fastest proliferating invention ever [16], quickly reaching a worldwide penetration that exceeds 60% [1]. Many of these devices are programmable mobile computers with advanced features, commonly referred to as *smartphones*. As these devices offer unprecedendent capabilities, users expect a new generation of applications to take advantage of them. Towards this direction, a lot of research is devoted on *context-aware, self-adaptive* applications. The latter improve users' experience by sensing their context and automating tasks that would otherwise require significant user attention.

This paper presents two variants of a context-aware application, built using the MUSIC development methodology and deployed on top of the MUSIC middleware [9]. The key focus is on describing the steps followed for the development of these applications, thus illustrating how this approach facilitates Separation of Concerns (SoC) and code reuse.

The paper is organized as follows: Section 2 introduces the foundations of the MUSIC context management framework, while section 3 shows how it can be used for the development of the *Context-aware Media Player* (CaMP). The experience of building the CaMP application is then analysed in section 4 and compared to related work in section 5. Finally, section 6 presents briefly the conclusions derived from this work.

2. THE MUSIC FRAMEWORK

As part of a more general framework [5, 15], the MUSIC context management framework provides developers a methodology for developing context-aware applications and a middleware platform for deploying them [9]. This methodology is closely coupled to the notion of *context plug-ins*: reusable pieces of code responsible for generating context data as needed. The middleware collects the generated data from the plug-ins and processes, stores and provides it to context listeners. As the whole middleware is built on top of the OSGi framework [19], the context plug-ins and the context management components are defined as OSGi bundles; i.e. the main artifact of reusable code in OSGi.

2.1 Context Plug-ins

Context plug-ins are the main components defined in the MUSIC context management system. They are classified into *context sensors* and *context reasoners*. The former are pure providers of context information, typically used as wrappers of physical hardware sensors (e.g. Bluetooth or GPS adapters), while the latter are more elaborate processors that take as input elementary context data and produce higher level context information; e.g. a User presence inferer or a WiFi signal strength predictor.

By leveraging OSGi's features, the context plug-ins are

*This research received partian financial support from the European Union (IST-MUSIC, 6th Framework Programme, contract number 35166 and ICT-AsTeRICS, 7th Framework Programme, contract number 247730).

designed as completely independent components, capable of enclosing necessary libraries as needed. For instance, a Bluetooth sensor may include Java libraries used to access the Bluetooth functionality as it is provided by the operating system. Moreover, a plug-in can even leverage native code (e.g., DLL libraries) when the underlying sensor requires native access to its resources (e.g., a face-tracking camera processor). In both cases, the OSGi specification provides adequate support for enclosing the functionality and the libraries in an OSGi bundle; i.e., a JAR file. This provides a clear-cut approach for searching, selecting and reusing context plug-ins from appropriate component repositories [9].

When deployed, the plug-ins effectively realize a *hierarchical processing chain*, where low-level context data is fed by sensors to reasoners; the latter are responsible to derive higher-level context information. This hierarchy forms a logical dependency graph, based on the provided and required context types of a component. At the top of the hierarchy are the context-aware applications that act as pure context consumers. However, since applications are not always active, the plug-ins are activated only when required. This led to an intelligent activation mechanism, which autonomously decides when each plug-in is activated based on its dependencies and requirements. This approach has experimentally shown a significant gain in resource consumption [11].

2.2 Context Model

As aforesaid, the context management middleware is responsible for controlling the plug-ins (i.e. managing their life-cycle), so as to collect and route the required context information and delegate it to the requested context clients as needed. Furthermore, the management middleware is responsible for managing the context data according to the predefined model. For this purpose, an elaborate Ontology-based context model was defined [13]. This model serves two important roles: First, it enables semantic consistency and, second, it enables context reasoning in terms of *relationships* among context entities as well as in terms of alternative *representations* used to encode the same context data.

In this work we are primarily concerned with the former: establishing semantic consistency. Since the proposed framework has code reusability as its primary goal, it is important to adopt a flexible model that enables semantic consistency (i.e., common dictionary) between various context providers and context consumers; potentially developed by various developers. For instance, a developer that needs to use the "location" context, must know what is the *key* needed to access it. Also, the model can serve by disambiguating the encoding used for abstracting the location information.

In the MUSIC model [13], the *keys* used to refer to context information have two components: First, the *scope* defines the context type; e.g. "#Thing.Concept.Scope.Location" refers to location. Next, the *entity* key disambiguates the entity to which the scope refers to. Thus, when the scope refers to a "location" context type, the entity is used to specify whether this type refers to a person or a device; e.g. the user is referred as "#Thing.Concept.Entity.User|myself", whereas "#Thing.Concept.Entity.Device|this" refers to the used device. This approach conforms with Dey's definition of context, which states that "*context is any information characterizing the interaction between a user and a device, including the user and the device themselves*" [3].

3. DEVELOPING CaMP

This section illustrates the design and implementation of a context-aware application using the MUSIC context framework. Two variations of the application are described: one suitable for deployment on a desktop device and another one for use on a mobile device. The goal of the process described is twofold: First, we reveal how the design and implementation of context-aware applications is simplified by leveraging existing or creating new context plug-ins. Second, we illustrate how these plug-ins can be reused in different settings.

The application itself consists of a normal media player, which features however some context-aware properties: First, it can detect when the user leaves his desktop—and when he gets back—to autonomously pause or resume the music playback. Second, it monitors the available bandwidth and switches between local and Internet-based streaming; default is Internet-based streaming.

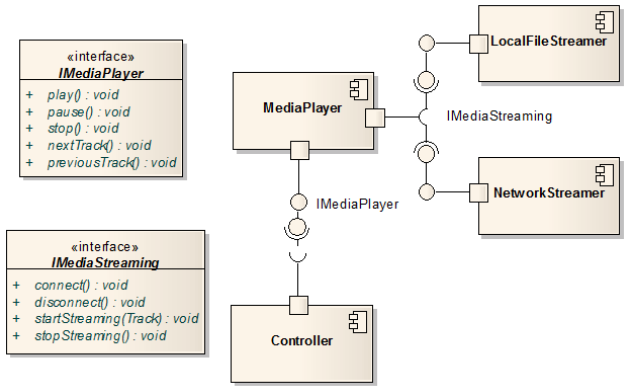

Figure 1: Architecture of CaMP

Figure 1 presents the application's core architecture, which defines its components and services. The components are:

- *Controller*: the main component of the application, realizing the UI and controlling the application state;
- *MediaPlayer*: provides functionality for media playback, as defined in the *IMediaPlayer* interface;
- *LocalFileStreamer*: supports streaming media from files stored locally on the device's storage system;
- *NetworkStreamer*: provides functionality for streaming media from web-based services.

The components interactions are characterized by their provided or required services, which are defined by the following interfaces:

- *IMediaPlayer*: Controls the state of the media player (e.g. pausing or resuming playback);
- *IMediaStreaming*: Handles generic media streamers by allowing to connect them and start or stop streaming.

Based on this architecture, we provide two variations of CaMP: one for desktop and another for mobile deployment.

3.1 Desktop-based CaMP

Initially, CaMP is built for deployment on a desktop computer. As such, it detects whether the user is sitting in front of his computer and pauses or resumes the media playback accordingly. In order to identify user's presence, the application leverages information from a "User presence reasoner" plug-in. In its turn, the reasoner uses input from two additional plug-ins: a "Bluetooth sensor" and a "Motion sensor".

The "Bluetooth sensor" connects to a hardware Bluetooth adapter and maintains a list of nearby Bluetooth devices. It is assumed that the user carries his phone with its adapter turned on in a mode allowing its discovery. Hence, the "User presence" sensor is able to infer whether the user is within communication range (e.g., < 5 meters) of his desktop.

However, the user might be within range but not necessarily in front of his computer. Thus, another sensor is utilized: the "Motion sensor". This sensor utilizes a web camera to periodically take pictures of the area in front of the computer. Thus, by comparing consecutive images it reports on the *movement activity* sensed in front of the desktop.

By combining the context data from the sensors, the "User presence reasoner" infers whether the user is sitting in front of his desktop. The derived context is leveraged by the *Controller* component to pause or resume playback accordingly.

In addition to the automatic control of the media player, the CaMP application features another context-aware property: it can automatically switch between local (file-system) and Internet-based streaming. The selection is decided solely based on the connectivity and the available bandwidth.

To realize this functionality, CaMP utilizes an additional context plug-in; i.e. "Connectivity sensor". This plug-in communicates directly with the underlying operating system and measures the available bandwidth in terms of Kilobytes.

The set of context dependencies defined for the application are illustrated in Figure 2. While dependencies are expressed in terms of *context types*, in practice each type corresponds to one of the above plug-ins, as well as their own dependencies. For instance, the "User presence reasoner" has a dependency on both the "Bluetooth sensor" and "Motion sensor". Finally, the "Connectivity sensor" has no dependencies.

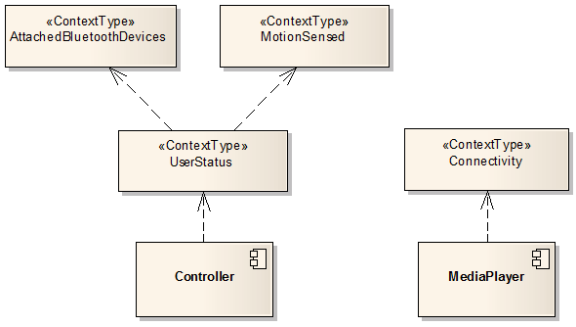

Figure 2: Context dependencies of Desktop CaMP

From a practical perspective, the actual binding of the application with the context management system is realized through an OSGi service published by the latter: the *context access* service [9]. This service provides the functionality for explicitly requesting values of a context type (i.e., synchronous access), as well as for subscribing for notification when new values are sensed for certain context types (i.e. asynchronous access). In addition to specifying simple queries, based merely on the requested context type, the clients can specify certain conditions used to filter the resulting context values as per the CQL dialect [14].

For example, the *Controller* could register for asynchronous notification to changes in the *user presence* as follows:

```
IEntity me = createEntity("#...Entity.User|myself");
IScope usrPrsn = createScope("#...Scope.UserPresence");
contextAccess.addContextListener(me, usrPrsn, this);
```

The *Controller* implements a specialized interface that defines a method for handling asynchronously communicated events. Internally, this method checks the encoded values (i.e. user presence: true or false) and acts accordingly (i.e. resumes or pauses playback). Similarly, the *MediaPlayer* registers for variations in the network bandwidth and selects the appropriate media streamer. Evidently, the application is only *loosely coupled* with the context management system (by means of context type dependencies), allowing for easy reuse of alternative realizations for each context provider.

3.2 Mobile CaMP

The Mobile CaMP is a variation of the Desktop CaMP, customized for deployment on smartphones. Thus, it does not provide automatic media start/stop, but provides a more sophisticated mechanism for selecting and adjusting the media streamer. The sophistication pertains the ability of the application to monitor the location and WiFi signal strength as the user moves about in space. Using this information a *prediction* is provided for the signal strength in the immediate future and exploited by the application for two purposes. First, select local file streaming when the network bandwidth is limited and, second, adjust the buffer size of the network-based streamer based on the predicted bandwidth.

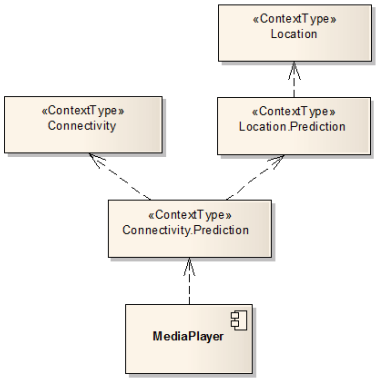

Figure 3: Context dependencies of Mobile CaMP

The functional architecture of the application remains the same as before; Figure 1. However, as illustrated in Figure 3, context dependencies have changed. In this case, the application—and in particular the *MediaPlayer* component—has a dependency on connectivity prediction. Hence, the "Connectivity prediction reasoner" has additional dependencies on network connectivity (i.e., current value of WiFi signal strength provided by the "Connectivity sensor") and on location prediction (i.e., projected location in the immediate future). The latter is provided by the "Location prediction sensor", which uses as input information concerning the current location, as provided by a "Location sensor".

While details on the implementation of the prediction

plug-ins is beyond the scope of this paper, it should be pointed out that a simple polynomial regression algorithm can be used. The algorithm takes the most recent user coordinates (as a function of time), in order to project an estimate for coordinates in a close future instant. More details on the context plug-ins implementation can be found in [9].

3.3 Deployment

The application's deployment is straightforward. Nevertheless, the user needs to ensure that the deployment platform includes the required context sensors (an obvious middleware limitation that could be overcome via a public plug-ins repository that allows discovery and automatic retrieval).

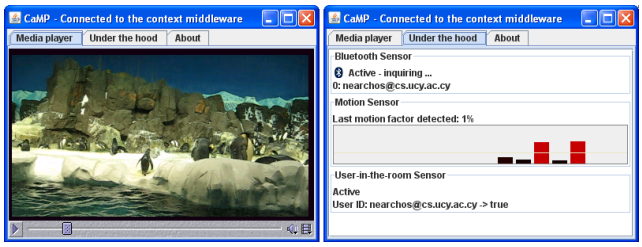

Figure 4: Screenshots of Desktop-based CaMP

For instance, the Desktop-based CaMP is deployed by ensuring that the required plug-ins (i.e. "User presence reasoner", "Bluetooth sensor", "Motion sensor" and "Connectivity sensor") are installed. Moreover, it is assumed that the context framework's bundles are deployed on the platform.

A screenshot of the application is illustrated in Figure 4. This application features multiple tabs. The first tab displays the media player's functional logic, which includes a video/audio player and controls for resuming/pausing playback, adjusting the volume, selecting tracks, etc. The second tab, is an *under-the-hood* view, which visually reveals the functionality of context plug-ins (i.e. for testing and demonstrating purposes). In this case, the three plug-ins used for identifying the user's presence are illustrated visually: The "Bluetooth sensor" provides a list with the IDs of all detected devices. The "Motion sensor" periodically captures and compares consecutive images, indicating their difference as a percentage (with those exceeding a threshold colored in red). Finally, the "User presence reasoner" combines information from the other two plug-ins and generates a true/false value when the predefined user (e.g. as indicated by their email) is present in the Bluetooth list while at the same time activity is sensed by the web-camera.

4. DEVELOPMENT EXPERIENCE

The previous section described two variations of the CaMP. Both variations use the same architecture for their functional logic (see Figure 1) but have different context dependencies and implement a different context-aware logic. It is argued that this approach promotes two important properties: *development with Separation of Concerns* and *Code reusability*.

4.1 Developing with Separation of Concerns

Arguably, Separation of Concerns (SoC) is an important method for relaxing the complexity which is inherent in modern software systems [8]. In effect, this method allows the developers to focus on different aspects of the software at a time, or even assign individual developers to work on each one of them in parallel.

The development method supported by the MUSIC context framework, facilitates such separation in two main ways:

- It separates the development of the functional aspects of an application (see Figure 1) from that of implementing its context-aware behaviour. The latter is partly implemented inside the plug-ins (e.g., WiFi signal strength prediction) and partly in the application (e.g., method that receives asynchronous notifications on registered context types and reacts accordingly).

- It separates the development of the context-aware behaviour in context producing and context consuming aspects. Hence, some context sensors act as pure context providers, while end-user applications act as pure context consumers. Occasionally, some context plug-ins serve both roles, where they are asked to handle low-level context data and infer higher-level context information, e.g. the "Connectivity prediction sensor".

4.2 Code reusability

Code reusability is closely related to SoC. In this work, reusability is enabled in terms of both functional and extra-functional logic [10]:

- Functional code can be reused in different applications, almost seamlessly. Leveraged by the SoC approach adopted in the developed applications, it is possible to port individual components of an application completely transparently—when the components do not include any context-awareness code; e.g., in the case of the two media streamers. This is in conformance to the component-oriented paradigm [18]. On the other hand, when ported components include context-aware logic that is concentrated inside a single method (see subsection 3.1), this code can be omitted altogether (e.g., see *Controller* component when its ported to Mobile CaMP) or edited as needed (e.g., see the *MediaPlayer* component when it is ported from the Desktop-based CaMP to the Mobile CaMP).

- The extra-functional code is fully reusable in terms of both the middleware infrastructure and the context plug-ins. The former refers to code supporting the plug-in lifecycle, the ontology model and components providing specialized functionality. Naturally, this is a common feature—and one of the main goals—of any middleware architecture. Conversely, reusability in terms of context plug-ins is a novel and handy feature, which enables context-aware functionality *out-of-the-box*. In the example of the two variants of CaMP, the "Connectivity sensor" is reused without change. Additionally, the available plug-ins can be further reused in completely different applications. For instance, an email client application could reuse the WiFi connectivity prediction in order to optimize its mail polling schedule. Finally, it should be noted that while the main advantage of using context plug-ins is their ability to facilitate code reuse at development time, it can extend even at run-time when multiple concurrently deployed applications share the plug-ins by requesting the same context types (e.g. the email client application described above).

5. RELATED WORK

Initial research on reusable code for context-aware applications was perfomed by Dey and Abowd [4], who stress out the lack of uniform support for developing and deploying context-aware applications. In particular, the authors denote that these types of applications are developed in an ad-hoc manner that is heavily influenced by the underlying implementation. Moreover, context independence is considered since developers should be able to build context-aware components separately from the application logic, so as to enable reuse of those components in miscellaneous applications. Hence, a Java-based Context Toolkit is defined that enables developers to built context-aware components (i.e., widgets, aggregators, interpreters) and use context information provided by them in order to develop different applications. Overall, the toolkit provides architectural support and delivers the generic mechanisms necessary to rapidly develop context-aware applications.

Another OSGi-based approach is the *Service-oriented, context-aware middleware* (SOCAM) [6]. Unlike our approach, SOCAM aims at offloading heavy context reasoning on residential gateways. However, because the individual context reasoning components (similar to the plug-ins of our approach) are directly bound to each other, component lifecycle is hindered and also combining context values from multiple sources becomes overly complex.

A more recent approach defined by Pokraev et al [12] proposes a Web Services platform for the rapid development and deployment of context-aware mobile applications. The platform is built with SoC in mind, since the platform's context-aware components are developed separately from the implementation of the application's logic. Consequently, each application is built on the basis of the context adaptation capabilities of the platform to provide a personalised, customised and dynamic user experience. In particular, the platform is tailored towards reuse of software components that are exposed as Web Services to the network to support the computing tasks of context-aware applications. Some examples of existing core components composing the platform are the Context Manager, Notification Manager, etc. Additional components are also implemented as third party services (e.g., context services), which interact with the core components to provide the context information and services required by applications.

Kapitsaki et al [7] define a Web Service architecture and a context adaptation process that supports the development and provision of context-aware applications. The architecture facilitates the execution of context management tasks that support the development of context-aware applications. In particular, applications are build by integrating software components (i.e., Web Services) and by utilising context plug-ins that manage context information. The approach aims towards reuse of the developed Web Service components and the context plug-ins, by keeping the logic of the context-aware application discrete from the context adaptation mechanisms, which can be independently plugged in. According to the authors, the main benefit of the approach lies in the complete separation of context adaptation tasks from both the application logic (i.e., combined Web Service components) and the thin client (i.e., web browser) of the application. Hence, the maintenance or modification of the context adaptation tasks is transparent from the client application, the Web Services and the user.

Finally, Serral et al [17] define a model-driven development method that allows defining context-aware applications as pervasive service models. These models allow describing both the functionality of the system and the context adaptation tasks. This means that once again the concepts of code reuse and SoC are taken into consideration but at a higher level of abstraction. In particular, a Pervasive system Modelling Language (PervML) is defined for specifying the system functionality in an abstract manner and a PervML ontology is defined for describing the concepts introduced in PervML. Consequently, from the defined PervML models the OSGi-based Java code and the OWL specification are generated using the corresponding transformation engines. The Java code implements the system's functionality and the OWL specification describes the system's context information used for adaptation-specific purposes. In overall, the approach allows reusing previously defined PervML models and automating the development of context-aware systems by transforming the models to the necessary artefacts.

5.1 Prototypes of Context-Aware Applications

One of the earliest context-aware application prototypes [4] was developed by Dey and Abowd: The *CybreMinder*, a context-aware system for supporting reminders. The prototype was developed using the Context Toolkit [2] and supports users in sending and receiving reminders. These reminders can be directly associated to contextual situations that involve time, place and other context types. The prototype includes three editors that are namely: the reminder creator, the context situation and the current reminders editor. First, the reminder editor allows defining reminders and, second, the situation editor enables dynamic construction of a rich contextual situation that is associated to the reminder. Finally, the current reminders editor displays a list of pending, completed, expired and delivered messages.

Pokraev et al developed the Context-aware Mobile Personal ASSistant (COMPASS) prototype [12] using the context-adaptation capabilities of their proposed platform. The prototype is a tourist guide that provides the user with information and services, which are of particular interest to the user given his current situation. For instance, a user that requires a place to spend the night is presented with a list of city hotels that match his accommodation preferences. Furthermore, the prototype's platform is open for third parties to integrate their information and services, so as application users can locate them and use them in a transparent manner.

The final prototype presented in this paper is a context-aware cinema application [7]. It integrates three web services: First, the *Greeting* service used to display a welcome message to the user based on his native language and current location, second the *ShowMovies* service that displays movies on the basis of the user's preferences and, third, the *Payment* service that selects the user preferred payment method. These services use the context adaptation capabilities provided by independently developed context plug-ins, which can be invoked when required using the platform's mechanisms; i.e., to automatically provide the context required by these services. The prototype demonstrates how it is possible to develop and use the context plug-ins without changing the application's logic. This is beneficial since the context plug-ins or the application logic can be independently modified or enhanced.

Unlike the discussed related work and the presented prototypes, CaMP offers a fully component-oriented solution, where the functional logic of the application (media streamer and player) is clearly separated from its extra-functional context-aware behavior. Arguably, this separation not only eases the development of the application but also it allows for quick prototyping of other applications (completely different, or derived as is the case of mobile/desktop CaMP) in a straightforward way.

6. CONCLUSIONS

In this paper we described the development steps required for creating two variants of the CaMP application. The proposed methodology provides significant benefits to developers of context-aware applications in terms of both Separation of Concerns—which eases development complexity and facilitates maintenance—and code reuse—in terms of reusable context plug-ins. Our experience has indicated that the proposed approach provides significant advantages in the hands of developers of context-aware applications targeting both mobile and ubiquitous computing settings. In the future, we aim at enabling further customization and a more elaborate lifecycle of plug-ins, as to enable for instance their finetuning in runtime depending on the needs of the applications that are using them.

7. REFERENCES

[1] *Measuring the Information Society: The ICT Development Index*. International Telecommunication Union, 2009.

[2] A. K. Dey. *Providing architectural support for building context-aware applications*. PhD thesis, Georgia Institute of Technology, 2000.

[3] A. K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[4] A. K. Dey and G. D. Abowd. The context toolkit: Aiding the development of context-aware applications. In *Proceedings of the ICSE Workshop on Software Engineering for Wearable and Pervasive Computing, International Conference on Software Engineering*, Limerick, Ireland, 2000.

[5] K. Geihs, R. Reichle, M. Wagner, and M. Khan. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 146–163. Springer Verlag, 2009.

[6] T. Gu, H. K. Pung, and D. Q. Zhang. Toward an OSGi-Based infrastructure for Context-Aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004.

[7] G. M. Kapitsaki, D. A. Kateros, and I. S. Venieris. Architecture for provision of context-aware web applications based on web services. In *Proceedings of the IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, Cannes, France, 2008.

[8] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *LNCS*, pages 195–213, Glasgow, Scotland, UK, 2005. Springer Verlag.

[9] N. Paspallis. *Middleware-based development of context-aware applications with reusable components*. PhD thesis, University of Cyprus, Sept. 2009.

[10] N. Paspallis and G. A. Papadopoulos. An approach for developing adaptive, mobile applications with separation of concerns. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)*, volume 1, pages 299–306, Chicago, USA, 2006. IEEE Computer Society Press.

[11] N. Paspallis, R. Rouvoy, P. Barone, G. A. Papadopoulos, F. Eliassen, and A. Mamelli. A pluggable and reconfigurable architecture for a context-aware enabling middleware system. In *Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications (DOA'08)*, volume 5331 of *LNCS*, pages 553–570, Monterrey, Mexico, 2008. Springer Verlag.

[12] S. Pokraev, J. Koolwaaij, M. V. Setten, P. D. C. T. Broens, M. Wibbels, P. Ebben, and P. Strating. Service platform for rapid development and deployment of context-aware mobile applications. In *Proceedings of the IEEE International Conference on Web Services*, pages 639–646, Florida, USA, 2005.

[13] R. Reichle, M. Wagner, M. Khan, K. Geihs, J. Lorenzo, M. Valla, C. Fra, N. Paspallis, and G. A. Papadopoulos. A comprehensive context modeling framework for pervasive computing systems. In *Proceedings of the 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08)*, volume 5053 of *LNCS*, pages 281–295, Oslo, Norway, 2008. Springer Verlag.

[14] R. Reichle, M. Wagner, M. U. Khan, K. Geihs, M. Valla, C. Fra, N. Paspallis, and G. A. Papa. A context query language for pervasive computing environments. In *Proceedings of the 5th IEEE Workshop on Context Modeling and Reasoning (CoMoRea'08) in conjunction with the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom'08)*, pages 434–440, Hong Kong, Mar. 2008. IEEE Computer Society.

[15] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*, pages 164–182. 2009.

[16] A. Rubin. The future of mobile, Sept. 2008.

[17] E. Serral, P. Valderasa, and V. Pelechano. Towards the model driven development of context-aware pervasive systems. *Journal of Pervasive and Mobile Computing*, 2009.

[18] C. Szyperski. Component technology - what, where, and how? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 684–693, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[19] A. L. Tavares and M. T. Valente. A gentle introduction to OSGi. *SIGSOFT Software Engineering Notes*, 33(5):1–5, 2008.