Dedicated Programming Support for Context-aware Ubiquitous Applications

Malte Appeltauer
Hasso-Plattner-Institut
Universität Potsdam
Germany
malte.appeltauer@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam
Germany
hirschfeld@hpi.uni-potsdam.de

Tobias Rho
Institut für Informatik
Universität Bonn
Germany
rho@cs.uni-bonn.de

Abstract

Ubiquitous mobile applications often require dynamic context information for user-specific computation. However, state-of-the-art platforms, frameworks, and programming languages used for developing such applications do not directly support context-dependent behavior with first class entities. Instead, context-aware functionality is tangled with the application's core concerns, which increases complexity, and hinders separation of concerns and further software evolution. This paper motivates Context-oriented Programming (COP) for ubiquitous computing. It presents an overview of our COP extension to the Java programming language and a scenario of a context-oriented mobile application.

1. Introduction

Rapidly evolving technologies for ubiquitous computing facilitate new application areas. For mobile applications, context information becomes an essential part. Context-aware applications are typically implemented in a service-oriented approach and composed by services, which provide context-dependent functionality. Location-based services, for example, are aware of the user's geographical position, health-and-fitness services monitor activity and vital signs such as heartbeat or blood pressure, while mood-based services take care of personal dispositions.

Due to the cross-cutting nature of context-dependent functionality, developers have to consider an additional dimension in the software model. Instead of representing context-dependent behavior separately at programming language level, *if* conditions check the presence of certain context information and thus tangle the code of the module's core concern with additional behavior. To cope with the complex task of developing context-aware applications, we need appropriate means for the representation of context-dependent behavior.

We propose an integral approach to the design, development, maintenance, and evolution of context-dependent services. *Context-oriented Programming* (COP) [4] enriches programming languages and execution environments with features to explicitly represent context-dependent behavior variations. In this paper, we motivate the need for a new language paradigm for the development of ubiquitous, context-aware applications and demonstrate, how we employ COP for this purpose. Throughout the paper, we exemplify our approach with the development of a mobile community platform, which is introduced in Section 2.1.

Section 2 introduces to context-aware application development and presents our scenario mentioned above. Section 3 describes the COP paradigm and motivates the development of ubiquitous, mobile applications using COP. Section 4 describes our approach for a COP extension to the Java programming language. Section 5 presents related work, while Section 6 concludes the paper.

2. Context-aware Service Development

Most context-dependent systems are developed based on context-management frameworks (e.g., [8, 1]). Such an infrastructure supports context reasoning, for instance, based on ontologies, and passes context information and change to applications. We do not focus on context-management frameworks but on enhancing the support of context-awareness at programming language level. Implementation issues of context-aware services are discussed in Section 2.2.

We will give an example for a service-based mobile application in Section 2.1. Since we want to point out the interaction of services and context information, we chose a domain with frequent context changes. We will refer this scenario throughout the paper and use it to exemplify context-oriented software development.

978-0-7695-3367-4/08 $25.00 © 2008 IEEE
DOI 10.1109/UBICOMM.2008.56

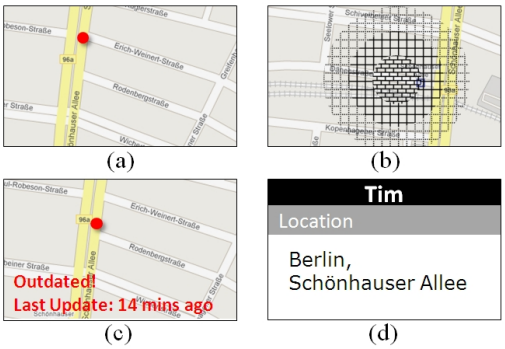

Figure 1. Different representations of Tim's location on Lucy's mobile device.

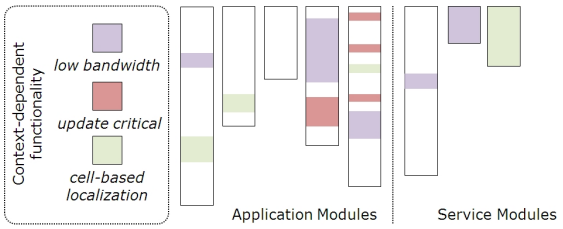

Figure 2. Context-dependent behavior as cross-cutting concern.

2.1. A Mobile Community Application

Mobile community applications (e.g., [8, 5]) allow users to share information about their mood, activities, location, and more. In this scenario, we focus on the location representation of buddies on mobile devices.

Two students, Lucy and Tim have an appointment at Hasso-Plattner-Institut (HPI) at 12:00. At 11:50, Lucy arrives at the campus. She checks her mobile device to inform herself about Tim's current whereabout. The graphical representation of this information depends on Lucy's context: If Lucy's device is currently connected to the Internet at high-bandwidth, a map image service is requested to render Tim's location on a map. Figure 1a gives an example for this map representation. Contrary, if the bandwidth is low, Tim's representation depends on Lucy's need for active information. This information is stored in Lucy's user profile and is accessible for applications. In the case that Lucy prefers a smart representation over the refresh period, a map with Tim's outdated location is shown (Figure 1c). The map is labeled with the time stamp of the last update. Alternatively, when Lucy insists on up-to-date information, while the bandwidth is too low for updating the map image, Tim's position data is simply shown as a text (Figure 1d).

At the same time, Tim arrives at the underground station, where his GPS device is unable to receive data any more. Thus, his mobile device switches to cell-based location detection. The new location data is calculated based on the position of the current mobile cell. The change of the location provider is propagated to Lucy's device. Depending on its current context, the device has to decide how it renders this additional location information. If the bandwidth is high, for instance, Tim's position is represented as a circular area with different color intensities indicating the probability of Tim's real location (Figure 1b).

2.2. No Explicit Context Representation

A system such as the community platform, which we describe in our scenario, can be developed with existing techniques for ubiquitous computing. Some mobile community projects provide functionality which is similar to our example (e.g., [8]). However, programming languages and environments do not provide appropriate abstractions for context-dependent behavior. Such functionality requires system adaptations at several points of the program. To give an example, Figure 2 sketches some modules belonging to our distributed application. The context-dependent functionalities concerning low bandwidth, the user's data update preferences, and localization data, are scattered over the client application and service modules. Because these *cross-cutting concerns* need to adapt the base modules at several points, they cannot be completely modularized within one object-oriented module. Instead, they tangle other modules with additional structure - for instance, *if*-conditions - at every possible point of behavior variation. The lack of an explicit representation hinders software maintenance and evolution.

Next to the modularization issues mentioned above, dynamic composition of application modules increase the complexity of development. In our example, the user rendering depends on dynamic context information. Figure 3 shows different renderings based on four behavioral variations. The figure does not consider alternative renderings when Tim's location data is based on cell IDs, which would double the number of possible adaptations. In general, with each new behavioral variation, the number of possible compositions increases, at worst exponentially. To control these different compositions, large condition constructs are injected into the system's structure at different points. Again, core functionality of modules is tangled with code that manages context-dependent behavior.

We see this tangling of application and context-management functionality as an indication for the need of language-level support of context representation and computation.

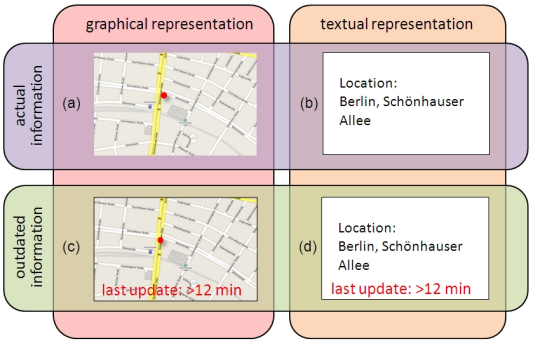

Figure 3. Behavioral variations based on layer compositions.

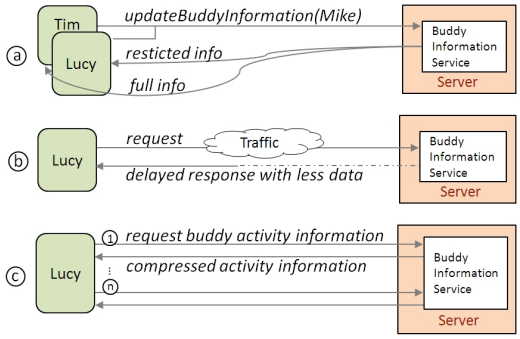

Figure 4. Behavioral variations in a service-oriented environment.

3. Context-oriented Programming

The context-oriented Programming paradigm [4] features a new dimension of software modularization by supporting an explicit representation of context-dependent functionality. Such functionality can be dynamically activated for a certain control flow and trigger dynamic behavior variations. With this dynamic composition of behavior variations, software evolution becomes more flexible and accessible.

We do not restrict the definition of context to a certain domain. A program's context could be everything within its execution and environment, such as personalization, sharing, location-awareness, activity, and more. We distinguish three basic classes of behavior variations:

Actor-dependent variations. A system might need to behave differently for different actors, even if the actors send the same request to the system. Consider, in our scenario Tim and Lucy regularly update the context information of their buddy Mike. When both request an update of Mike's activity information, while Lucy's is connected at low bandwidth, the service responses differently to the clients. Tim receives the full information update. Lucy, due to the low bandwidth, receives just a restricted set of data, see Figure 4(a).

Environment-dependent variations. Variations, which are based on external events and are not given implicitly by the control flow, are denoted as environment-dependent. Figure 4(b) represents an environment-specific variation: Our platform delays its service response when network traffic increases.

System-dependent variations. A system may alter its behavior, for instance, depending on its state and history. When Tim and Lucy frequently update information of their buddies, the system could alter its response depending on the information sent out before. Then, it would be unnecessary to submit information about all buddies at once. Instead, updates could be distributed on successive responses. This system-specific variation is shown in 4(c). The response of `updateBuddyInformation()` contains only a subset of all buddy information, i.e., the most outdated one. This buddy list differs for every request.

For a broad introduction to COP we refer to [4]. COP has been successfully implemented in the form of several language extensions, such as ContextL for Lisp, ContextS for Squeak/Smalltalk, ContextPy and PyContext for Python, and ContextR for Ruby. In the following section, we sketch ContextJ.

4. ContextJ

4.1. Language Features

In this section, we describe the main features of *ContextJ*, our COP extension to the Java programming language. We chose Java as base language for the implementation of our scenario mainly for two reasons: Firstly, Java is a widely accepted language proven many times to be a stable and secure platform for service-oriented programming. Secondly, Java would be the first statically typed programming language to be extended with context-oriented features. We want to investigate, how the dynamic properties of COP are feasible for statically typed languages. ContextJ provides means for the definition and dynamic activation of behavioral variations:

Layer. A `layer` encapsulates partial method declarations belonging to a behavioral variation. Similar to annotations, layers enrich source code with additional semantic informations.

Partial method declarations. Partial methods are defined within layers. They contain instructions which are executed - while the layer is active - before, after or instead of the original method execution. The function `proceed()` can be used within partial declarations. It dispatches the method

invocation to the next layer - if more layers are active - or to the original declaration. This feature is also implemented by other languages such as Common Lisp and AspectJ.

Layer activation. Layers can be activated and deactivated, depending on the current context at run-time. When activated, method calls are dispatched to the layered methods. The construct `with(`*Layer*`)` controls activation.

4.2. Implementation

The development of a ContextJ compiler is work in progress. A first prototype, *ContextJ** [4], has been developed previously as a proof of concept. ContextJ* is a plain Java library and covers a limited set of COP features.

The listings presented in this paper are implemented with a second prototype, based on a generic aspect library, developed with *LogicAJ*[1] [7]. The aspect implementation is oblivious for developers and needs no adaptation for concrete applications. To use the aspect's COP functionality, some idioms need to be followed when writing Java programs. Layered methods are declared with the first parameter to be a subtype of `Layer`. The ContextJ `with` construct is simulated with the methods `activateLayer(Layer)` and `deactivateLayer(Layer)`. Additionally, the aspect library provides a static `proceed()` method which executes the dispatching to the next layer or to the original method.

With the development of COP for Java with LogicAJ we gathered experiences which will help us to create a complete Java language extension.

4.3. Example

In the following, we present a COP-based implementation of parts of our scenario. We focus on (1) the modularization of context-dependent behavior variation with layers and (2) the dynamic composition of layers.

Modularization. Figure 5 shows two classes belonging to an implementation of the application introduced in Section 2.1. The class `Client` controls the application on the mobile device; `BuddyActivityService` on the server provides the client with new buddy activity information. To extend our application with the behavioral variations of our scenario, we need to adapt our system both, on the client and server sides. The following listing presents a layered method declaration of `getBuddyActivity()`.

```
public static Activities getBuddyActivity(String userId){
  return getUser(userId).getActivity();
}
public static Activities getBuddyActivity(
                TextualRepresentation tr, String userId){
  return filterImages(proceed(userId));
}
```

[1] LogicAJ and the Java/COP aspect library are available at: http://sewiki.iai.uni-bonn.de/research/logicaj/cop

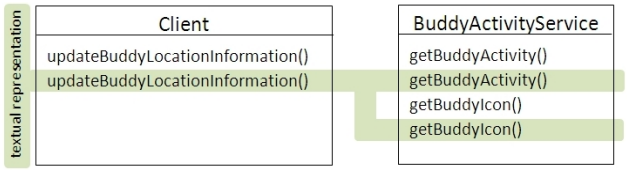

Figure 5. Distributed layer declaration

While the first declaration in the listing is a plain Java method declaration, the second one represents a partial declaration of that method for the layer `TextualRepresentation`. The method uses the `proceed()` function, which delegates the call to the original method. The return value is wrapped with `filterImages(Activities)`, which filters potential graphical data out of the activity list. When the `TextualRepresentation` layer is active, only this restricted set is passed to the client.

Dynamic Composition. Consider, the functionalities for the different location representations shown in Figure 3 are modularized into different layers. Each combination of these layers yield to a different configuration. The dynamic configurations are controlled by dynamic layer activations. The next listing shows such an activation.

```
Activity a;
if(getBandwidth().isLow()){
  activateLayer(OutdatedInformation.class);
  a = client.requestService("getBuddyActivity","Mike");
  deactivateLayer(OutdatedInformation.class);
}
else{
  a = client.requestService("getBuddyActivity","Mike");
}
```

The service call `getBuddyActivity()` is invoked within the *low bandwidth* context, which is inferred by the method call `getBandwidth().isLow()`. Thus, Lucy receives the compressed list of activities computed by the layered method declaration of the previous listing. The layer activation is thread-local; a parallel service request of Tim, for instance, would not be dispatched to the layered method.

Combinations of layer activations lead to different compositions. The following listing shows a layered method `renderUser()` which is responsible for the client's graphical representation of a user.

```
public UIComponent renderUser(ActualInformation ai){
 String loc = requestService("getBuddyLocation","Tim");
 return renderMap(requestService("getMap",loc));
}
public UIComponent renderUser(OutdatedInformation oi){
 UIComponent com = proceed();
 com.add(renderTimeStampOfLastUpdate());
 return com;
}
public UIComponent renderUser(TextualRepresentation tr){
 String loc = requestService("getBuddyLocation","Tim");
 return renderText(loc);
}
```

The listing below contains an activation of multiple layers to compose the behavior shown in Figure 3(d).

```
activateLayer(OutdatedInformation.class,
              TextualRepresentation.class);
renderUser();
deactivateLayer(OutdatedInformation.class,
                TextualRepresentation.class);
```

The invocation of `renderUser()` is passed to the `OutdatedInformation` layer. First, the `proceed()` function delegates the call to the next layer. The `TextualRepresentation` layer renders the location simply as a text and returns the component back to `OutdatedInformation` which adds a time stamp to it.

4.4. A COP-based Service Infrastructure

Since we want to validate the features of ContextJ for service development in a realistic environment, we are constructing a service-oriented infrastructure. It is based on several different client systems and server platforms, in which we integrate our application of Section 2.1. The application contains services implemented with ContextJ. Our infrastructure is based on IYOUIT and Android.

IYOUIT [8] is a mobile community platform. Users can share context information, such as their current activity, location, mood, weather information, and more with their friends. Next to context data derived by the system, users can create new context, for instance blog entries and photos, which are tagged with the author's context information. The implementation of our scenario is based on IYOUIT. We extend the system with new features implemented with ContextJ. Later on, we will additionally implement services with ContextS for the Smalltalk-based Seaside [11] server. With this heterogeneous infrastructure we will evaluate COP.

The Open Handset Alliance project *Android* [10] is an open-source platform for mobile devices. It provides a Java run-time environment for application development. Location-based services, such as GPS, are directly supported by Android. However, it does not come with a context-management system. Since IYOUIT only supports a client software for Nokia S60 cell phones, we develop a second IYOUIT client for the Android operating system.

5. Related Work

5.1. Context-Management Frameworks

Context-dependent systems are mostly endured with a context-management framework (CMF). The CMF manages context-reasoning and processing and provides context information to applications. We already discussed IYOUIT as one example for CMF-based systems.

The *Java Context-awareness Framework* (JCAF) is a service-based infrastructure for context-aware applications. In this framework, context acquisition, management and distribution is implemented with different services that interact with each other. Due to the service-oriented approach, JCAF-based applications are flexible for extension, even at run-time.

The Context Toolkit [13] influenced many of nowadays CMFs. It incorporates various services related to the gathering and supply of context, including the encapsulation of context, access to context data, and a distributed infrastructure. Several approaches have been developed in the tradition of the Context Toolkit with improvements regarding e.g. personalization and end-user development [17] or ontology-based context modeling [3]. All share the mentioned limitation.

5.2. Aspect-oriented Programming

The purpose of Aspect-oriented Programming (AOP) is the encapsulation of cross-cutting concerns. The core concept of AOP languages is a *join point model*, well-defined points in the execution of a program, interceptable by *advice* constructs which are executed before, after or around these points. The join points, where an advice will be executed, are selected via a (declarative) *pointcut language*. All terms have been coined by AspectJ [6] and have become standard. The advice constructs encapsulate a concern in an aspect that would otherwise be scattered throughout the code. Typical examples for cross-cutting concerns are persistence, caching, and logging. In context-aware applications, context-dependent concerns can be extracted into aspects and (de-)activated for certain contexts.

Context-aware applications mostly run on mobile devices using distributed services. Some AOP languages and systems come with explicit support for distributed AOP, such as *AWED* [9] and *ReflexD* [16]. The approaches can be facilitated for context-aware service selection and decoration (filtering). For a feature comparison of distributed AOP approaches, see [2]. However, the classic join point models have no explicit representation for context. They therefore need additional frameworks for context-management and analysis, which reduces the advantage of the declarative pointcut language, since the location where to apply an aspect can not be described in one place.

An open framework for *Context-aware Aspects* based on a reflective extension of Java [15] is proposed by [14]. Aspects can be restricted to certain contexts, refer to context parameters in advice, and via a snapshot approach to previous context states.

The OSGi-based aspect language *CSLogicAJ* [12] is an extension to LogicAJ [7]. It features the adaptation of service behavior by context-sensitive service aspects. Con-

text change can be modeled as a join point of the system. Context join points can be described within the language's pointcut constructs, which are based on first-order logic. Asynchronous advice executes service adaptation triggered by context changes. CSLogicAJ is based on the OSGi-based middleware *Ditrios* [12]. Only services available in the local OSGi registry can be intercepted.

6. Conclusion

Some platforms for mobile applications support context-awareness via context-management systems at application level, but with poor support at programming language level. Hence, the implementation of context-dependent concerns tangles the application's core modules and hinders software maintenance and evolution.

In this paper, we propose to apply the COP paradigm to support the development of context-aware services. We present a scenario of a mobile application which is implemented using ContextJ, a COP extension to the Java programming language. We demonstrate, how software modularization can be enhanced with layers and dynamically scoped layer activation and composition.

The explicit representation of behavioral variations at programming language level improves software modularization and supports the development of context-aware systems.

Acknowledgments

The authors thank Michael Haupt, Robert Krahn, Jens Lincke, and Daniel Speicher for valuable discussions and comments on drafts of this paper.

References

[1] J. Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In *Proceedings of the 3rd International Conference on Pervasive Computing*, Albrecht Schmidt, University of Munich, May 2005.

[2] F. Dantas, T. Batista, and N. Cacho. Towards Aspect-Oriented Programming for Context-Aware Systems: A Comparative Study. In *SEPCASE '07: Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[3] T. Gu, H. K. Pung, and D. Q. Zhang. Toward an OSGi-based Infrastructure for Context-Aware Applications. In *IEEE Pervasive Computing, vol. 03, no. 4*, Oct-Dec 2004.

[4] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[5] R. Kernchen, D. Bonnefoy, A. Battestini, B. Mrohs, M. Wagner, and M. Klemettinen. Context-awareness in MobiLife. In *15th IST Mobile & Wireless Communication Summit*, Mykonos, Greece, 2006.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[7] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects. In *RAM-SE'04: Workshop on Reflection, AOP, and Meta-Data for Software Evolution, in conjunction with ECOOP'06*, pages 111–126, Oslo, Norway, June 15 2004. Fakultät für Informatik, Universität Magdeburg.

[8] J. Koolwaaij, A. Tarlano, M. Luther, P. Nurmi, B. Mrohs, A. Battestini, and R. Vaidya. Context Watcher – Sharing context information in everyday life. In J. Yao, editor, *Web Technologies, Applications, and Services*, Calgary, Canada, July 17-18 2006. ACTA Press, Proceedings of The IASTED International Conference on Web Technologies, Applications, and services.

[9] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM.

[10] Open Handset Alliance. Android. http://www.code.google.com/android.

[11] M. Perscheid, D. Tibbe, M. Beck, S. Berger, P. Osburg, J. Eastman, M. Haupt, and R. Hirschfeld. *An Introduction to Seaside*. Software Architecture Group, Hasso-Plattner-Institut, April 2008.

[12] T. Rho, M. Schmatz, and A. B. Cremers. Towards Context-Sensitive Service Aspects. In *Workshop on Object Technology for Ambient Intelligence and Pervasive Computing, co-located with ECOOP 06*, Nantes, France, July 3-7 2006.

[13] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM.

[14] E. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-Aware Aspects. In *Proceedings of the 5th International Symposium on Software Composition*, Lecture Notes in Computer Science. Springer, March 2006.

[15] É. Tanter and J. Noyé. A Versatile Kernel for Multi-language AOP. In R. Glück and M. R. Lowry, editors, *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, September 29 - October 1 2005.

[16] É. Tanter and R. Toledo. A Versatile Kernel for Distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, Jun 2006. Springer-Verlag.

[17] A. Zimmermann, M. Specht, and A. Lorenz. Personalization and context management. *User Modeling and User-Adapted Interaction*, 15(3-4):275–302, 2005.