Context-oriented Programming for Mobile Devices: JCop on Android

Christopher Schuster
Department of Computer Science
University of California, Davis, USA
cschuster@ucdavis.edu

Malte Appeltauer Robert Hirschfeld
Software Architecture Group
Hasso-Plattner-Institut, Germany
{first.last}@hpi.uni-potsdam.de

ABSTRACT

The behavior of mobile applications is particularly affected by their execution context, such as location and state a the mobile device. Among other approaches, context-oriented programming can help to achieve context-dependent behavior without sacrificing modularity or adhering to a certain framework or library by enabling fine-grained adaptation of default behavior per control-flow.

However, context information relevant for mobile applications is mostly defined by external events and sensor data rather than by code and control flow. To accommodate this, the JCop language provides a more declarative approach by pointcut-like adaptation rules.

In this paper, we explain how we applied JCop to the development of Android applications for which we extended the language semantics for static contexts and modified the compiler. Additionally, we outline the successful implementation of a simple, proof-of-concept mobile application using our approach and report on promising early evaluation results.

Keywords

Mobile applications, context-oriented programming, dynamic adaption

1. INTRODUCTION

A recent trend in mobile computing is to equip user devices with more and more computational capabilities, graphics acceleration, memory capacity and sensors. These advancements in hardware are accompanied by the rise of increasingly sophisticated mobile applications. Most of these applications are either explicitly or implicitly affected by the context and state of the mobile device and/or the user. Examples for the former case are location-based applications which provide the user with directions or other kinds of services that are closely related to the user's current location. The latter case is true for all applications depending on an Internet connection to deliver a service to the user.

Besides location and Internet availability, there are other factors that a mobile application programmer has to take into account in order to adapt the application's behavior, such as battery status, current time, uplink and downlink bandwidth, mobility, sound volume, screen brightness, the user's language, preferences and his or her intent. There are various ways for the programmer to address these issues and implement context-sensitive behavior. However, most approaches so far have drawbacks regarding the application architecture and modularity because they require the programmer to structure the code in a certain way which is tied to the context-oriented middleware used. Section 5.2 lists some of these alternatives.

Instead of providing a framework or an API for the programmer to deal with the context, we present in this paper a more declarative approach at programming language level that uses context-oriented programming (COP) and pointcut-based activation of adaptations which is less invading and more modularized than frameworks and libraries. Therefore, we applied the Java-based JCop language [2] to the Android system. Android's architecture required modifications in both the language definition and the compiler's byte code generation.

Section 2 introduces the Android platform and the JCop language. Section 3 describes our approach and briefly outlines its implementation including some practical issues. Early performance results and a short evaluation is then given in Section 4. Related work is summarized in Section 5. Finally, as this implementation is a first step into a new direction, there are many ideas and opportunities for improvement as explained in Section 6, which also gives some closing remarks.

2. FOUNDATIONS AND MOTIVATION

2.1 The JCop Language

The JCop language [2] extends Java with COP constructs. It allows for the explicit representation of execution context-dependent *behavioral variations.* These variations are encapsulated into *partial methods* that are defined in layers - special objects that can dynamically control method dispatch. To change the dispatch of a method `m` from its default implementation to one of its variations, the corresponding layer can be activated for the control flow of a specific execution. Layer activation can be defined either explicitly (by surrounding a method call with a *with block*), or declarative (using *pointcuts* and *context objects*).

```
 1 public class PhoneCall {
 2   public void onIncomingCall() {
 3     ...
 4     ring();
 5   }
 6 }
 7
 8 public layer SilentMode {
 9   public void PhoneCall.onIncomingCall() {
10     vibrate();
11   }
12 }
13
14 public context ClassRoom {
15   when(enteredClassroom()) :
16     with(SilentMode);
17
18   private boolean enteredClassRoom()  {
19     // check GPS location
20   }
21 }
```

Listing 1: JCop example using explicit layer activation.

The example in Listing 1 shows a usage of these constructs on a phone application. By default, the phone rings on incoming phone call (Line 4). However, in some contexts (e.g., on entering a a classroom), this basic behavior should be replaced by silent mode vibration. The layer `SilentMode` provides a partial method for `onIncomingCall` that replaces the ringing with vibration (Lines 8–12). Layer activation is controlled by the `ClassRoom` context object. It contains a declarative activation of the layer (Lines 15–16) and an auxiliary method that accesses sensors to determine the current location (Lines 18–20). A thorough presentation of JCop can be found in previous work [2].

2.2 Android

Android is a Java-based software stack for mobile devices including an operating system, middleware, such as telephony, location, and notification managers, a mobile application framework and core applications, such as a Web browser and text messaging. The Android application framework offers, among others, access to the device hardware and location information; both can be used to gather context information. Android applications are instances of one of four component types: single screen applications (activities), background processes (services), content providers or event listeners (broadcast receivers). Android does not use a standard Java virtual machine (VM) but a register-based VM called *Dalvik VM* that has been optimized specifically for mobile devices.

2.3 Challenges for a JCop-Android Integration

Programming Android applications involves adhering to the coding guidelines and using a special compiler to generate Dalvik bytecode. Both of these restrictions affect the way COP can be applied to Android.

In order to run a mobile application on Android, the code has to implement specific interfaces, declare specific permissions and callbacks, subclass specific framework components and override certain methods. Additionally, the application programmer has no direct control over the main event loop

```
public class DownloadEntryTask ... {
  public Entry doIt(...) {
    client = AndroidHttpClient.newInstance("");
    Entry e = loadEntry(main);
    loadPicture(e, main);
    return e;
  }
}
```

Listing 2: Straight-forward implemenation in Java

and has to keep a separation between code that is to be executed by a non-blocking graphical user interface thread and a blocking background computation thread. This makes thread-level activation of contexts as normally used with JCop difficult. Every callback could possibly be executed by another, unrelated thread which means that the control flow is not the best criteria for context activation. An alternative way will be presented in Section 3.

Furthermore, the Dalvik VM has certain restrictions that are relevant for implementing COP. First of all, it currently does not support custom classloaders, dynamic code generation and bytecode manipulation at runtime. This means that a COP solution can only use static weaving and cannot rely on any of these dynamic features for context activation/deactivation. Secondly, the Davlik VM does not use normal stack-based Java bytecode, but instead expects special bytecode generated from compiled Java class files by the `dx` tool. Theoretically, the `dx` tool can also convert bytecode that was not generated by `javac` but it makes certain assumptions that might not be always true, e.g. that the generated method call instruction avoids the virtual method table for constructors and private method calls.

3. APPLYING JCOP TO ANDROID

3.1 The APOD Mobile Application

The best way to describe the approach is by proving a simple example Android application and the way it was implemented by using COP. Suppose a mobile device user is interested in science and wants an application that automatically downloads and displays the current Astronomy Picture of the Day (APOD) as published by NASA alongside a short descriptive text. This application was implemented for the Android platform as a simple graphical user interface that asynchronously downloads the current APOD from the Web. The simplified code for a straight-forward implementation is shown in Listing 2.

This version is not context-aware which means that it would just crash without an error message if Web access is not available. An improved version would first check the online/offline state of the device and then display an offline entry, e.g. by using a conditional `if` branch. Even better would be a solution that downloads APOD in a high resolution if and only if the downlink bandwidth is large enough, e.g. when the mobile device uses WiFi. There are many more extensions that would improve the user experience by being more context-oriented. However, a naive implementation would just use many conditional branches which can lead to duplicate code and unreadable code especially when multiple, unrelated context-sensitive variables have to be taken into account.

```
public layer OfflineEntry {
  public Entry DownloadEntryTask
                        .loadEntry(Context ctx) {
    return new Entry("No␣network␣available");
  }
}
public static context NetworkContext {
  when (!Network.connected()) :
    with (OfflineEntry);
}
```

Listing 3: JCop solution

By using COP language constructs, the code becomes more modular, but the layer activation itself can usually not be located in the application code since the relevant parameters of the context like the current state or location of the mobile device are managed independently from the application and can change at any time. Therefore, we used the pointcut language of JCop as described in Section 2.1. It provides the advantage of declarative layer activation and deactivation based on a boolean expression. By using a static method for accessing the current network state and a layer to adapt the download entry, a network-sensitive solution for the example application would be implemented as shown in Listing 3.

3.2 Static Contexts

There is one important difference between this solution and the code displayed in Listing 1. As mentioned in Section 2.3, the application programmer's control over the threads and callbacks in Android is limited. This means that even with automatic layer activation and deactivation, the application code gets executed by different threads with different currently active contexts.

One solution for the example would be to manually activate the `NetworkContext` each time the control flow reaches the application code, e.g. at the beginning of each callback method. This can easily be done by providing the application programmer with special Android subclasses that have guaranteed context activation. However, this approach would be very similar to the existing framework approaches for context-handling as presented in Section 5 and too intrusive for the application programmer.

Instead, we introduced 'static' contexts which are always active and potentially affect every running thread in the system. This is specifically aimed at cases like the one given in Listing 3 where the context is not connected with certain code fragments or control flows within the code, but instead serves as a global context-dependent behavioral adaption affecting the whole application. This new semantic for contexts was then implemented by extending the JCop language with an optional `static` Java modifier for JCop `context` declarations. This only affects one application instance because applications in Android are separated into processes with each process having its own virtual machine. However, it is noteworthy that this model is primarily targeted at coarse behavior adaption based on a global, relatively stable context because changes in the layer activation of the static context will not affect currently executing layered methods after the condition clause was evaluated.

3.3 Generating Correct Dalvik Bytecode

Apart from the language extension, the code generation of JCop was also changed in order to get the example application running on an Android device. One modification involved changing the flags of the generated class file while another one dealt with an issue of the `invokevirtual` Java bytecode instruction. As mentioned in Section 2.3, the Dalvik VM expects private method calls to circumvent the virtual method table, because these methods cannot be overridden in subclasses and can therefore use link-time binding. In contrast to the standard Java VM, these methods do not even have entries in the virtual method table which results in the `invokevirtual` instruction jumping to another method than was originally intended. This restriction can either be met by changing the code generation of JCop (which internally uses AspectJ) or by treating private methods as protected after the first compilation. For the prototype, the latter method was chosen due to simplicity.

4. FIRST RESULTS

4.1 Applicability

With the adjustments described in Section 3, the example Android application was successfully compiled and installed on an Android device. The current, early version of JCop for Android has support for COP based on the network connection state which was sufficient for implementing the example application. If the phone was connected by WiFi, then the high resolution APOD was downloaded; if there was no Internet connection, then a placeholder was displayed together with an error message; in all other cases, the normal download code is executed. Due to the way JCop is implemented, this condition is checked at every layered method call that is affected by the active set of layers. This might lead to multiple checks in a very short amount of time. To avoid getting this information repeatedly from the operating system, which, depending on the operating system's caching strategy, might involve reading the sensor values from hardware at a unnecessary high rate, the device-related context variables are cached for a specific time, for example 1 second in the current JCop implementation for Android. Alternatively, it would also be possible to subscribe to the Android OS for events relevant to the application and update the cached context variables accordingly but this is an implementation detail.

Additional context-dependent variations can simply be implemented by adding a layer definition with partial methods and another rule in the context definition describing the condition and the list of layers. The rest of the application code does not have to be modified which shows how COP can be used to increase modularity in mobile applications.

4.2 Micro Benchmarks

For evaluating the approach, not only the functional properties of JCop for Android, but also non-functional properties like performance and memory consumption have to be considered. Therefore, we conducted micro-benchmarks to compare JCop/Android method dispatch throughput with plain Java. JCop's layer-aware method dispatch implemented based on its predecessor ContextJ [1]. The compiler transforms layers and partial methods into plain classes and forwarding methods. Given a list of active layers L, execution of a layered method $o.m$ will first lookup the (thread-

Approach	Runtime
No context-dependency	2901ms
Conditional `if` branching	2959ms
JCop on Android	3450ms

Table 1: Measured runtime performance

local) layer list for the first layer providing a suitable partial method. If no such method exists, the base method (that is located in o) is called. If a partial method exists, the call is dispatched to the layer that in turn calls its partial method implementation (located in o).

In the example application, no difference in runtime performance could be measured. However, in the example there is only one context-sensitive method call which is executed only once per program execution. To get more meaningful results we set up a micro-benchmark running on a virtual Android 2.3.1 device inside the Android Emulator on an Intel Core Duo processor with 1.66 GHz running a Linux 2.6.35 kernel. For reference, the first context-insensitive executes 1000 calls to method A followed by 1000 calls to method B. The second implementation executes a loop with 2000 iterations, each time deciding whether to call A or B based on a state variable. The third result then uses JCop for Android to execute 2000 context-dependent method calls to A with 1000 of these calls layered by B[1].

The results of this benchmark are given in Table 1 and suggest that, in this simple case, the context-dependent method dispatch has a runtime performance comparable to that of a straight-forward implementation using conditional branches. This similarity in runtime reflects the mechanism used by JCop to achieve context-dependent behavior. However, these results might be different for systems with more layers and more context definitions because these might introduce overhead for checking additional conditions.

5. RELATED WORK

In the following, we discuss related work from the language design and context management perspective.

5.1 COP Languages for Java

The first ideas about a COP extension to Java have been presented [6] to improve the accessibility of the *ContextL* code discussed in that paper. Core COP functionality has been implemented by the language prototypes *ContextJ** [8] and *ContextLogicAJ* [3] and by the ContextJ [1] extension that supports language constructs and concrete syntax for layer declaration within classes and explicit layer composition. All three approaches only provide explicit layer activation and no means for declarative composition rules as JCop does.

The EventCJ [9] language is closely related to JCop. Much like JCop, it makes use of pointcuts and events for layer activation. However, layer activation and event handling own a slightly different semantics. Declarative layer activation is defined by transition rules whereas JCop expects an explicit list of layers to be (de)activated at a join point.

[1]The benchmark source code is available at
`https://www.hpi.uni-potsdam.de/swa/trac/Cop/attachment/wiki/JCop4Android/jcop4android_benchmark_v0.3.zip`

5.2 Context Management Systems

Du and Wang presented a solution [7] that also helps the programmer with context-aware code on mobile devices. Their solution invokes methods at each state transition. These transitions can be seen as the entrance and exit of different contexts and are defined in an XML file. A context is defined by one or more ranges of sensor values. Some drawbacks of this solution are, that it requires the programmer to achieve context-awareness solely by the use of callbacks, that the context has to be representable by a set of sensor values[2] and that these sensors are read at a certain interval even when the application does not need the context to be updated. JCop for Android solves this problem by reading sensor values only on demand and no more than once per second.

There are also context-oriented middleware that have a different focus than COP and therefore do not provide the right granularity in defining the control-dependent behavior, e.g. *Aura* [4] is more task-oriented and does not provide context-awareness for just a few lines of code. Solutions like *CARMEN* [5] on the other side, are using policies and profiles to change the behavior of an application depending on the context. This is also used to mitigate between applications that are competing for the resources of the mobile device and therefore the application programmer has no full and unrestricted control over context rules defined by the system or the user.

Android itself provides support for handling the context. Besides accessing the mobile device state synchronously, it is also possible to subscribe to certain events that announce a change of the context, e.g. by using the `PhoneStateListener`, certain callbacks are called on each state change. This makes it possible to develop context-aware applications but requires either manually accessing the context at each decision point or structure the code so that context changes cause state changes in the application. Both of these strategies are adequate for small and simple applications but can become tedious in case of large, heavily context-sensitive applications.

6. FUTURE WORK

The performance evaluation discussed in Section 4.2 are promising but certainly a little bit too simplistic. A more thorough benchmark has to be used in order to evaluate the performance in more realistic scenarios with a larger codebase and more context-dependent behavioral adaptations, but that is beyond the scope of this paper.

Furthermore, the claims of increased modularity by COP in comparison with framework/library approaches are not backed by measurable numbers. Future work could focus on the code quality and could use the means of code metrics like the cyclomatic complexity, class cohesion or simply lines of code.

The Android example in Section 3 only dealt with a simple contextual variable, the network connection state. Other parameters like sensor data and especially the GPS location as part of the context are an important research topic with many opportunities for applying COP in a declarative manner.

[2]Additionally, not all relevant context parameters can naturally represented as numbers, e.g. the user's preferences or intents.

As mentioned in Section 4, the results of device-related parameters are cached for one second to avoid redundant sensor readings. Depending on the application, the sensor and the intent of the programmer, this caching duration might be too long or not long enough. Future versions of JCop for Android should be more flexible in that regard and finding the right sensor check interval would be an interesting topic for future work.

Regarding the static contexts introduced in Section 3.2, the presented language extension basically allows the scope of a context to be the whole application instance rather than a specific control flow. A more general approach would allow the deployment of contexts for other scopes as well.

In conclusion, the context of the user and the mobile device plays a very important role for mobile applications. COP offers the right means to efficiently and elegantly implement context-awareness to deal with the dynamically changing context and to improve the user experience. The solution presented in this paper demonstrates how JCop was applied to Android in order to enable COP for mobile devices. The early results presented in this paper are very promising and should be further investigated.

7. REFERENCES

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ - Context-oriented Programming for Java. *Computer Software of The Japan Society for Software Science and Technology*, 28(1):1_272–1_292, 2011.

[2] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the 9th International Conference on Software Composition*, Lecture Notes in Computer Science, pages 50–65, Berlin, Heidelberg, Germany, 2010. Springer-Verlag.

[3] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. Dedicated Programming Support for Context-aware Ubiquitous Applications. In *UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 38–43, Washington, DC, USA, 2008. IEEE Computer Society Press.

[4] Matthias Baldauf and Schahram Dustdar. A Survey on Context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, page 2004, 2004.

[5] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware middleware for resource management in the wireless Internet. *Software Engineering, IEEE Transactions on*, 29(12):1086 – 1099, 2003.

[6] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103, Berlin, Heidelberg, Germany, September 19 2006. Springer-Verlag.

[7] Weichang Du and Lei Wang. Context-aware application programming for mobile devices. In *Proceedings of the 2008 C3S2E conference*, C3S2E '08, pages 215–227, New York, NY, USA, 2008. ACM.

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[9] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj: a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 253–264, New York, NY, USA, 2011. ACM.