Design and Development of a Context Oriented Language for Middleware Based Applications

Andrea Sindico
Elettronica S.p.A. Italy
andrea.sindico@elt.it

Giovanni Bartolomeo
DIE – Dept of Electronic Engineering
University of Rome "Tor Vergata"
Italy
giovanni.bartolomeo@uniroma2.it

Vincenzo Grassi
University of Rome "Tor Vergata"
Italy
vgrassi@info.uniroma2.it

Stefano Salsano
DIE – Dept of Electronic Engineering
University of Rome "Tor Vergata"
Italy
stefano.salsano@uniroma2.it

Abstract

Nowadays context-aware adaptation is becoming an important feature for pervasive computing applications. In this paper we present JCOOL, a COntext Oriented Language tailored to handle context awareness in Java applications. JCOOL exploits Aspect Oriented techniques so that context changes detection and related adaptations can be considered as two separated crosscutting concerns with respect to the core "business logic" of new or legacy Java applications. Moreover, mobile and pervasive applications generally rely on middlewares that hide the complexity of the underlying environment. In order to show how JCOOL support can be introduced into middleware based application, in the second part of the paper we also describe JCOOL integration in SMILE [1], a Middleware Independent Layer developed in the scope of the SMS project [2].

Categories and Subject Descriptors: D.3.2, D3.3 [**Language Classifications, Language Constructs and Features**]: Specialized application languages – Frameworks.

General Terms: Design, Languages.

Keywords: context awareness, aspect oriented programming, domain specific language, middleware.

1. Introduction

Specific mechanisms and API are needed to support context dependent modifications of the behavior of mobile and distributed applications. Existing platforms that try to achieve this goal using general-purpose languages (GPLs), suffer from the common difficulties of GPLs related to the lack of semantic expressiveness of their constructs. Besides, the adaptation to different contexts can be considered as an orthogonal task with respect to the core application logic [3]. In this respect, Object Oriented GPLs suffer from their inability to encapsulate crosscutting concerns, such context awareness, without affecting the components business logic. This suggests the adoption of a Context Oriented Programming approach based on the use of Domain Specific Languages (DSLs) tailored for the context awareness needs: these languages can better capture the crosscutting nature of context awareness and provide more effective constructs to aid the developer in tackling this concern.

This paper describes an ongoing work on the definition of a context oriented language named JCOOL (Java COntext Oriented Language) we have recently started to design and develop as a follow up of the work made in [4]. One of the main goals of JCOOL is the possibility of introducing context awareness capabilities into an already existing Java application without changing its original code. To show how this can be achieved we propose an example of JCOOL integration into SMILE [5][6], a "Simple Middleware Independent LayEr" between applications and the underlying middleware platform. The goal of SMILE is to relieve the developer from the need of writing middleware specific code, focusing instead on the implementation of the application business logic.

2. Related Works

Context-oriented Programming (COP) is a new programming approach which aims to alleviate the spreading of context-dependent behaviours throughout a program by incorporating context as a first-class construct of a programming language [7][8][9]. In [10][11] the following list of mechanisms a Context Oriented Programming Language should provide is described:

- *Behavioral variations*: variations tipically consist of new or modified behaviour of the system components;
- *Layers*: Layers group related context-dependent behavioural varations;
- *Activation*: Layers aggregating context-dependent behavioural variations can be activated and deactivated dynamically at runtime. Code can decide to enable or disable layers of aggregate behavioural variations based on the current context;

Workshop NAOMI 2008,1st April 2008, Brussels, Belgium.

- *Context*: Any information which is computationally accessible may be part of the context upon which behavioural variations depend;
- *Scoping*: The scope where layers are activated or deactivated can be controlled explicitly.

Aspect Oriented Programming (AOP) [12] can be exploited to address these requirements. For example, Layers of behavioural variations can be realized by the definition of ad hoc *around* advices whose activation is triggered by other Aspects which play the role of Context monitors as explained in [4]. However, AOP languages only consider the elementary events in the execution flow of a program such as method calls, field accesses, and so on, which in AOP terminology are called *join points*. AOP join points are not expressive enough to cover the complexity of a context definition which instead may depend on complex and distributed properties of the system components and even of its execution environment. In [3], E. Tanter et al. point out that AOP languages are also limited with respect to the kind of context dependencies that can be expressed. For example, even though there are a number of AOP languages that make it possible to define pointcuts which depend on past execution history, because of the lack of an explicit context definition, they only consider simple events such as method invocations but don't consider past contexts.
Tanter et al. also propose a list of characteristics a context definition should have:

- *Stateful*: a context may have state associated with it;
- *Composable*: different context definitions can be combined to define complex contexts;
- *Parametrized*: context can be defined generically, and parametrizied by aspects that are restricted to it;

In [3], a Reflex extension is proposed which addresses the above requirements. Reflex itself is a Java extension which provides building blocks for facilitating the implementation of different aspect oriented languages so that it is easier to experiment with new AOP concepts and languages. In the framework described in [3] the developer has to define how and when a context has to be saved so that it will be possible to refer to it in a future instant. In this respect, JCOOL makes it easier to refer to past contexts. For example the developer does not have to define how context must be saved because the only informations a context should provide, except his state, are the actual parameters which has verified it. How these parameters are stored is hidden to the developer by the JCOOL underlying environment.
In [13] Costanza et al. describe ContextL, a Context Oriented Programming Language for Common Lisp Object System, which provides a set of language constructs that allow the developer to associate partial class and method definitions with layers. Layers can then be activated and deactivated in the control flow of a running program. When a layer is activated, the partial definitions become part of the program until this layer is deactivated.
The main difference between ContextL and JCOOL is that ContextL does not provide language constructs to define a Context and its inner states. ContextL only provides a macro, named `with-active-layers`, to activate a layer in the dynamic scope of a program. However, even though the developer does not have to spread context adaptation code in the base program, which is instead encapsulated in the layer definitions, it has to spread `with-active-layers` block of code in the base program in those points where the related context changes. JCOOL provides distinct language constructs for *Context Monitoring* and *Context Adaptation*. It considers these two concerns as two crosscutting concerns: the former crosscuts the base program to detect when it is in a context of interest, the latter crosscuts the Context Monitors to introduce context adaptations when needed. In this way we achieve a strong separation between *when* and *how* context adaptation should be carried out. Moreover, these two concerns are well encapsulated in two distinct first-class language constructs. Thanks to this, the base program is not affected by any of these two concerns. Moreover, because of the lack of an explicit context definition, ContextL does not address the Context definition requirements proposed in [3], which are instead explicitly taken into account by JCOOL.

3. JCOOL

JCOOL is a domain specific aspect oriented language derived from the UML Profile for context awareness described in [4]. In JCOOL there are two main constructs named `Context` and `Adaptor` that are the code level counterparts of the *ContextMonitor* and *ContextAdaptor* elements defined in the aforementioned UML profile. As its UML equivalent, a `Context` is composed by a set of rules which specify conditions that must hold to introduce some kind of context adaptation.
In JCOOL each `Context` is identified by a unique name and can involve one or more components of the base system. This means that a `Context` definition can affect only those classes of the base system that are listed after the key word `involve`.
A `Context` is represented as a state machine with a default start state and one ore more states in which it may migrate. To this end, a state transition rule is associated to each state. The relation between a state and its transition rule is expressed with an Horn Clause in a Prolog-like syntax [14].

```
stateName :- stateTransitionRule
```

A `Context` is in a given state until the related transition rule holds while it is in the default state if none of its transition rules is verified. A state transition rule consists of a set of one or more predicates, over the components involved by the Context, combined with the logical operators ","(AND), "|" (OR) and "!" (NOT). Because of these characteristics JCOOL's Contexts can be considered *Statefull* thus addressing the first requirement defined in [3].

```
Context SimpleContext involve ClassA {

    default;
    stateA    :-(ClassA *.attribute == 1 );
    stateB(i) :-(ClassA i.attribute == 2 );

}

Context ComplexContext involve ClassB,
                               SimpleContext

{
    default;
    compositeState:-(SimpleContext.stateB),
                    (ClassB *.attributeB >1);

    complexState:-[(SimpleContext.stateA)+,
                  (SimpleContext.stateB){2}],
                   (SimpleContext.stateA);

}
```

Figure 1. Examples of Contexts

Contexts are also *Composable*, because they may be built as a composition of other contexts. The states of a composite Context have state transition rules that depend on the state transitions of the Contexts it is composed by.
Figure 1 depicts two examples of Context definition. The first one consists of a Context named SimpleContext which involves the ClassA class of an hypothetical base system. This context can be in two different states: stateA and stateB, depending on the value of an attribute of a ClassA's instance.
The second Context depicted in Figure 1, named ComplexContext, is an example of composite context because it depends on the SimpleContext context and on the ClassB class. It starts in the default state but migrates in the compositeState state as soon as the SimpleContext is in the stateA state and the value of an attribute of a ClassB's instance is greater than a certain value.
Sometimes it could be necessary to detect a precise sequence of events in order to consider a Context in certain state. To this end, square brackets must be used to enclose those events of a state transition rule that must occur in the exact sequence they are written in. Operators ?, + and * can be used, like in regular expression, to express that an event should occur respectively: never or one time; at least one time; never or any time. Curly brackets can be used to enclose the exact number of times an event must occur. In composite context this syntax can be used to define a context state which depends on an exact sequence of past contexts and possibly refers to their context parameters. For example, the context ComplexContext of Figure 1 migrates in the complexState only after that the SimpleContext has migrated into the stateA at least one time, then it has migrated in the stateB two times and it is currently in the stateA.
On the transition between two states, a context may trigger the execution of one or more Adaptors through the invocation of one of its entry points (Figure 2). As its UML counterpart, an Adaptor is a container for context adaptation mechanisms. Each Adaptor is identified by a unique name and may be driven by one or more Contexts, as well each Context may drive several Adaptors. Parameters can be passed to the adaptation action after a transition rule is evaluated and fired. These parameters can be free variables which take the values of those objects which verify the fired state transition rule. For example, the i variable used in the SimpleContext.stateB definition, takes the value of the ClassA instance which verifies the related state transition rule when fired.
An Adaptor has as many entry points as the state transitions it is designed to intercept. For each entry point two kinds of adaptation can be defined: *one shot activities* and *behavioural variations*.
One shot activities consist of two pieces of code associated to an Adaptor's entry point: the former must be executed at the related context-state incoming event (in); the latter has to be executed at the related context-state outgoing event (out). Within these blocks it is possible to use the optional parameters passed with the state transition.
Behavioral variations, or layers, consist of a set of alternative method definitions that may affect classes or particular class instances passed as parameters to the Adaptor. A behavioural variation is active until the involved Context remains in the related state. When a behavioural variation is no longer active the methods it has affected return to their original implementations.
The difference between these two kinds of adaptation is that one shot activities are executed as soon as the related Context goes in/out a certain state. They can use objects passed by the related context to perform activities preparatory to the context change. Behavioural changes, instead, have to be considered as a dynamic override of some methods of objects or classes of the base system

```
Adaptor SimpleAdaptor {

    SimpleContext.stateA {

        //stateA incoming adaptations
        in:{
         System.out.println("coming in the
         SimpleContext.stateA context state");

        }

        //stateA outgouing adaptations
        out:{
         System.out.println("Going out from the
          SimpleContext.stateA context state");
        }

        //stateA layers (Behavioral changes)
        public void ClassA.simpleMethod(){
          System.out.println("Alternative method
                              Implementation");
        }


    }

    SimpleContext.stateB(i){

        //stateB incoming adaptations
        in: {...}
        //stateB outgoing adaptations
        out: {...}
    }

}
```

Figure 2. Example of Adaptor

that change their behaviour until they remain in a certain context state. As mentioned, behavioural changes may affect classes or instances so that, in a given time, different objects of the same class may have different implementations of the same methods depending on their context. When a behavioural variation is removed the methods it has affected return to their original implementations.
Figure 2 depicts an example of Adaptor which is driven by the SimpleContext of Figure 1. As soon as the SimpleContext enters in the stateA the in block of code of the related SimpleAdaptor entry point is executed. Until the SimpleContext is in that state, behavioural changes are introduced that consist, for this example, in the overriding of the ClassA.simpleMethod method. Since this behavioural variations is related to the ClassA it affects all the ClassA instance of the system. When the SimpleContext goes out the stateA the out block of code of the related SimpleAdaptor entry point is executed and the behavioural variations are deactivated so that the ClassA.simpleMethod returns to its original implementation.

4. SMILE

Developed in the scope of the SMS Project [2] SMILE is an abstract platform [15] with the explicit goal of avoiding developers to rewrite their applications as a consequence of changes in the underlying middleware, allowing to focus the development effort on the business logic more than on the implementation details. An application written for SMILE consists of a set of peers, named SMILEPeers, which are abstract classes loose coupled with the underlying runtime environment.

From the developer point of view, SMILE peers are autonomous entities which may communicate through message exchanges. Each peer may access a common minimun set of features provided in form of an API. These features typically include naming and addressing, service registry, message routing mechanisms, etc. and are implemented by exploiting the underlying middleware facilitations.

In order to exploit these facilitations, without directly relying on them, SMILE provides a mechanisms called *binding*, similar to the one defined in Web Service Description Language (WSDL) [16]. Thanks to this separation layer, applications written for SMILE are not to be changed as a consequence of changes in the underlying middleware platform; instead only the binding has to change. Unlike WSDL, however, the same SMILE application, at run-time, might exploit more than one binding, thus dynamically adapting its behaviour to different contexts. More details can be found in [5][6].

In the following section we will describe how this feature represents an interesting use case for JCOOL, which may be seamlessly integrated into SMILE. For clearness' sake, with the help of an example, we will describe step by step the procedure developers have to follow to successfully achieve such an integration, together with some internal details the platform hides them, in order to properly run JCOOL context oriented applications in a seamless way.

5. JCOOL as COP Support for SMILE

Developers wishing to use JCOOL support in SMILE first have to identify possible join points; in addition to application specific operations, these include specific pointcuts provided by the SMILE API, which are of four kinds: callbacks for implementing the application lifecycle; methods to interact with the service registry; methods and callbacks for message exchanges and for remote procedure calls; interfaces between the applications and the bindings. Subsequently, developers add two additional sets of files to their SMILE application source code: one set defining `Context`, with initial state and state transition rules; the second set defining the `Adaptors`. Both these file sets have a global scope, i.e. they can refer to any object (including custom objects) defined in the sources.

As an example, consider a SMILE application composed by a set of SMILE Peers with some of them having to send messages requiring a high privacy level. JCOOL can be used to introduce a context aware adaptation so that, depending on the reliability of the transport protocol available in the currently active binding, a SMILE Peer should send or not its message.

Figure 3 depicts the definition of a JCOOL Context named `MediumReliability` which involves the SMILEPeer class. Suppose this context can be in two different states: *low* and *high*, depending on the security level provided by the transport protocol used by a given binding.

The instance parameter, used in the context definition, is a formal parameter which is evaluated whenever the state transition rule is fired. Once evaluated, it is passed as actual parameter to any Adaptor triggered by this Context. In this example, whenever the `MediumReliability` context migrates into the low state, it triggers the execution of `PrivacyAdaptor`. `PrivacyAdaptor` prints out a message to alert about the context change and introduce a behavioral variation that changes the send method of the passed `SMILEPeer` instance so that this instance will not send any message requiring a high privacy level. Note that this change affects only `SMILEPeer` instances which are using an unsecure binding whereas other `SMILEPeers` continue to use seamlessly their original implementation of the send method.

The SMILE platform takes care of implementing such a logic seamlessly , in two simple steps. At compile time, JCOOL Adaptors pass through an ad hoc pre-processor that weaves them with legacy sources in order to insert adaptation code. At runtime, an entity called "Broker", implementing inversion of control and listening at any event related to peers contained in a given platform instance, is responsible also to monitor the bindings to the underlying middleware platforms. Whenever the application uses JCOOL support, context states and transition rules contained in a JCOOL Context are dynamically interpreted by the Broker which finally invokes the execution of triggered adaptation actions whenever needed.

```
Context MediumReliability involve SmilePeer{

  default;
  low(instance):-instance.currentBinding.
                          equals(SipBinding);
  high(instance):-instance.currentBinding.
                  equals(XMPPBinding) &&
        ((XMPPBinding)instance.currentBinding).
        getTransport().
        equals(XMPPBinding.HTTPS));

}

Adaptor PrivacyAdaptor {

  MediumReliability.low(instance) {
    in : {
        System.out.println("Warning unsecured
                            medium ");
    }
    out : { //No action }
    //layer definition
    Public void instance.send(Message msg){
     if((msg.getOverallPrivacyLevel()==HIGH){
        msg.getSender().printMsg("Message
        privacy level not compliant with the
        current medium");
     } else { proceed(); }
    }

  }
                                  }
```

Figure 3. Examples of JCOOL in SMILE

6. Conclusions

The ultimate goal of research in Context Oriented Programming is to provide language constructs to aid software developers in a better encapsulation of crosscutting context dependent behaviors.

In this paper we have presented JCOOL, a domain specific language that makes possible a strong separation between the *Context Monitoring* and *Context Adaptation* concerns with respect to the base system. This feature aids the designer to think at these two concerns separately, designing different *Context Monitors* and *Adaptors* that can even be reused and combined into different architectures to achieve the desired degree of context awareness. Moreover, in order to show how JCOOL support can be provided into a middleware for distributed applications, we have also described JCOOL integration into SMILE.

What we have presented is a first step of an ongoing work. In the future, we intend to investigate about the possibility to import Prolog knowledge bases in a Context definition so that its state transition rules, thank to their horn clause syntax, may use *modus ponens* to detect inferred context states. For example: if the fact "Paris is in France" is known, the context `location("Paris")` will be accepted as a match of the context `location("France")` [9].

Another open issue concerns the coincidental activation of different behavioural variations that affect common target components; i.e. different behavioural variations that affect the same methods of the same components at the same time. Currently, for each component, behavioural variations affecting the same method are activated in a stack like way so that when a behavioural variation is activated it automatically deactivates the previous one. However we would investigate about a better way to solve this issue i.e. by providing a way to automatically merge in a unique variation independent not conflictual variations that affect the same components.

We are currently working on the development of a first prototype of JCOOL pre-processor that will perform a static weaving of Context and Adaptors' code with a given target base system. This first goal will not cover the possibility to handle unforeseen context adaptation. However, to address the issue of unpredictable context changes and related adaptations, we are already investigating about the possibility to exploit runtime weaving capabilities of modern AOP environment [17].

References

[1] *The SMILE (Simple Middleware Independet LayEr) Project*. http://netgroup.uniroma2.it/twiki/bin/view.cgi/Main/SmilePublic

[2] *The IST-Simple Mobile Service (IST-SMS) Project*. http://www.ist-sms-org

[3] E. Tanter, K. Gybels, M. Denker, A. Bergel *Context Aware Aspects*. SC 2006 (W. Lowe, M. Sudholt eds), LNCS 4089, 2006 pp.227.242

[4] V. Grassi, A. Sindico *Towards Model Driven Design of Service Based Context Aware Applications*. International Workshop on Engineering of software services for pervasive environments. ISBN:978-1-59593-798-8, pages 69-74, ACM, USA.

[5] G. Bartolomeo, S. Salsano, R. Glaschick, *A Glimpes into SMILE Programming*. http://netgroup.uniroma2.it/twiki/bin/viewfile.cgi/Main/SmilePublic?rev=1;filename=tr-smile-v1.0.pdf

[6] S. Salsano, G. Bartolomeo, C. Trubiani, N. Blefari Melazzi: *SMILE, a Simple Middleware Independent LayEr for distributed mobile applications*. IEEE WCNC 2008, March 31-April 1, 2008, Las Vegas

[7] M. Gassanenko, *Context Oriented Programming: Evolution of Vocabularies.* Proceedings of the euroFORTH'93 Conference. Marianske Lazne, Czech Republic.

[8] M. Gassenenko, Context Oriented Programming. euroFORTH'98, Schloss Dagstuhl, Germany.

[9] R. Keays, A. Rakotonirainy. *Context Oriented Programming*. International Workshop on Data Engineering for Wireless and Mobile Access, San Diego, USA, 2003. ACM Press.

[10] R. Hirschfeld, *Modularizing Context-dependent Behavioral Var ations with Context-oriented Programming*. Generative and Trans formational Techniques in Software Engineering, Braga, Portugal 2007.

[11] R. Hirschfeld, P. Costanza, O. Nierstrasz. Context-oriented Pro gramming. In Journal of Object Technology (JOT), vol. 7, no. 3, pages 125-151, March-April 2008, www.jot.fm;

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M Loingtier, J. Irwin, *Aspect Oriented Programmin*: In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Eds Vo. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 220-252.

[13] P.Costanza R. Hirschfeld *Language constructs for context oriented programming*. In Proceedings of the ACM Dynamic language Symposium, San Diego, California, USA, October 18,2005 ACM DL..

[14] W.F. Clocksin, C.s. Mellish, *Programming in Prolog*, Springer Verlag New York, Inc., New York, NY, USA, 1987.

[15] P. A. Almeida, *Model-Driven Design of Distributed Applications*. Ph.D. (TI/FRS/018), The Netherlands, 2006, ISBN 90-75176-422

[16] W3C *Web Services Description Language (WSDL) 1.1*. W3C Note 15 March 2001, http://www.w3.org/TR/wsdl

[17] W. Vanderperren, D. Suvèe, B. Verheecke, M. Agustina Cibràn, V. Jonckers. *Adaptive Programming in JAsCo*. In Proceedings of AOSD 2005, ACM Press, Chicago, USA.