Wolfgang De Meuter (Ed.)

LNCS 4406

Advances in Smalltalk

14th International Smalltalk Conference, ISC 2006
Prague, Czech Republic, September 2006
Revised Selected Papers

Springer

Wolfgang De Meuter (Ed.)

Advances in Smalltalk

14th International Smalltalk Conference, ISC 2006
Prague, Czech Republic, September 4-8, 2006
Revised Selected Papers

Springer

Volume Editor

Wolfgang De Meuter
Programming Technology Laboratory
Vrije Universiteit Brussel
Belgium
E-mail: wdmeuter@vub.ac.be

Library of Congress Control Number: 2007923851

CR Subject Classification (1998): D.1, D.2, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-71835-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71835-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12044788 06/3180 5 4 3 2 1 0

Preface

The 14th International Smalltalk Conference took place in the first week of September 2006 in Prague, Czech Republic. This volume contains the peer-reviewed technical papers that were presented during the academic track of the conference.

The International Smalltalk Conference evolved out of the annual meeting of the European Smalltalk User Group (ESUG). This meeting usually lasts about a week and allows Smalltalk experts to discuss Smalltalk solutions and environments. The meeting attracts a diverse audience consisting of Smalltalkers from industry as well as from academia. Thanks to the perpetual effort of people like Stéphane Ducasse, Noury Bouraqadi, Serge Stinckwich and Roel Wuyts, over the years the ESUG meeting was provided with a separate academic research track during which researchers could present academic results about Smalltalk and its development tools. Unfortunately, no formal publication forum was associated with this track, which made it less attractive for authors to submit a paper. Starting with this edition of the conference, we hope this will change. An agreement was reached with Springer to publish a post-conference proceedings of this 14th edition. I think our community owes a big thank you to Stéphane for this! Hopefully next year this agreement can evolve into a 15th edition of the conference with formally announced proceedings. This will certainly motivate more Smalltalk researchers to submit a paper!

The conference accepted just over half of the submissions. Although this can be interpreted as a sign of low quality, I think it is not. The set of researchers conducting their research in Smalltalk is quite small. However, as is the case with the code produced by Smalltalkers, the quality-to-quantity ratio of the research is high. This is confirmed by the fact that all papers were reviewed by at least three members of the international Program Committee. The committee consisted of a number of researchers that are highly renowned in the field of object-oriented programming in general and in the Smalltalk world in particular. I would like to thank them for their efforts in trying to make this a conference of outstanding quality.

September 2006 Wolfgang De Meuter

Organization

Program Committee

Table of Contents

Application-Specific Models and Pointcuts Using a Logic Meta Language

Johan Brichau[2,3,*], Andy Kellens[1,**], Kris Gybels[1], Kim Mens[2], Robert Hirschfeld[4], and Theo D'Hondt[1]

[1] Programming Technology Lab
Vrije Universiteit Brussel, Belgium
{akellens,kris.gybels,tjdhondt}@vub.ac.be
[2] Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
{johan.brichau,kim.mens}@uclouvain.be
[3] Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille, France
[4] Hasso-Plattner-Institut
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract. In contemporary aspect-oriented languages, pointcuts are usually specified directly in terms of the structure of the source code. The definition of such low-level pointcuts requires aspect developers to have a profound understanding of the entire application's implementation and often leads to complex, fragile, and hard to maintain pointcut definitions. To resolve these issues, we present an aspect-oriented programming system that features a logic-based pointcut language that is open such that it can be extended with application-specific pointcut predicates. These predicates define an application-specific model that serves as a contract that base-program developers provide and aspect developers can depend upon. As a result, pointcuts can be specified in terms of this more high-level model of the application which confines all intricate implementation details that are otherwise exposed in the pointcut definitions themselves.

1 Introduction

Aspect-oriented Software Development (AOSD) is a recent, yet established development paradigm that enhances existing development paradigms with advanced encapsulation and modularisation capabilities [1,2]. In particular, aspect-oriented programming languages provide a new kind of abstraction, called *aspect*, that allows a developer to modularise the implementation of crosscutting concerns such

* This work was partially supported by the European Network of Excellence AOSD-Europe.

** Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

as synchronisation, transaction management, exception handling, etc. Such concerns are traditionally spread across various modules in the implementation, causing tangled and scattered code [3]. The improved modularity and separation of concerns [4], that can be achieved using aspects, intends not only to aid initial development, but also to allow developers to better manage software complexity, evolution and reuse.

One of the most essential characteristics of an aspect-oriented programming language is that aspects are not *explicitly* invoked but instead, are *implicitly* invoked [5]. This has also been referred to as the 'obliviousness' property of aspect orientation [6]. It means that the *base program* (i.e., the program without the aspects) does not explicitly invoke the aspects because the aspects themselves specify when and where they need to be invoked by means of a *pointcut definition*. A pointcut essentially specifies a set of *join points*, which are specific points in the base program where the aspect will be invoked implicitly. Such a pointcut definition typically relies on structural and behavioural properties of the base program to express the intended join points. For example, if an aspect must be triggered at the instantiation of each new object of a particular class, its pointcut must capture those join points whose properties correspond with the execution of the constructor method. As a result, each time the constructor method is executed (i.e. an instance is created), the aspect is invoked. In most aspect languages, this corresponds to the execution of an *advice*, which is a sequence of instructions executed before, after or around the execution of the join point.

Unfortunately, in many cases, defining and maintaining an appropriate pointcut is a rather complex activity. First of all, an aspect developer must carefully analyse and understand the structure of the entire application and the properties shared by all intended join points in particular. Some of these properties can be directly tied to abstractions that are available in the programming language but other properties are based on programming conventions such as naming schemes. 'Object instantiation' join points, for example, can be identified as the execution of constructor methods in languages such as Java. Accessing methods, however, can be identified only if the developers adhere to a particular naming scheme, such as through `put-` and `get-` prefixes in the method names. In contrast, a language such as C# again facilitates the identification of such accessor method join points because they are part of the language structure through the C# 'properties' language feature. In essence, we can say that the more structure is available in the implementation, the more properties are available for the definition of pointcuts, effectively facilitating their definition. However, structure that originates from programming conventions rather than language structure is usually not explicitly tied to a property that is available for use in a pointcut definition. This is especially problematic in languages with very few structural elements such as Smalltalk. In such languages, application development typically relies heavily on the use of programming conventions for the implementation of particular concepts such as accessors, constructors and many more application-specific concepts. As a result, aspect developers are forced to explicitly encode these conventions in pointcut expressions, often resulting in complex, fragile, and hard to maintain pointcut expressions.

The aspect-oriented programming language that is presented in this paper features an *open*, *logic-based* pointcut mechanism that allows to tie structural implementation conventions to explicit properties available for use in pointcut definitions. Our approach builds upon previous work on logic-based pointcut languages where we have described how the essential language features of a logic language render it into an adequate pointcut definition language [7]. In this paper, we further exploit the full power of the logic programming language for the definition of application-specific properties. In particular, we present an integration of the AspectS [8] and CARMA [9] aspect languages for Smalltalk. The result is an aspect-oriented programming language in which pointcuts can be defined in terms of an application-specific model that is asserted over the program. The application-specific model captures the structural conventions that are adhered to by the developers of the program and reifies them as explicit properties available for use in pointcut expressions. The model as well as the pointcuts are implemented using logic metaprograms in SOUL [10].

In the following section, we present AspectSOUL, the integration of the AspectS and CARMA aspect languages. Next, in section 3, we implement a number of pointcuts that rely on typical structural conventions that are adhered to by application developers in a Smalltalk environment. We explain how such pointcuts are complex, fragile, and hard to maintain and, in section 4, we describe how our AspectSOUL allows to tackle these issues through the definition of application-specific pointcuts, expressed in terms of an application-specific model. Section 5 applies the approach to aspects that operate on the drag and drop infrastructure of the UI framework and the refactoring browser in the Smalltalk environment. We summarize related and future work in section 6 before concluding the paper.

2 AspectSOUL

AspectSOUL is an integration of the CARMA pointcut language [9] and AspectS [8], a Smalltalk extension for aspect-oriented programming. Unlike most other approaches to aspect-oriented programming, AspectS does not extend the Smalltalk programming language with new language constructs for writing down aspects and advice expressions. Instead, AspectS is a framework approach to AOP. Pointcuts are written as Smalltalk expressions that return a collection of joinpoint descriptors. CARMA on the other hand, is a dedicated pointcut language based on logic programming. Naturally, such a dedicated query language offers advantages for writing pointcuts, as pointcuts are essentially queries over a joinpoint database. The integration of this logic-based pointcut language with AspectS further enforces the framework nature of AspectS by providing a full-fledged query-based pointcut language that can be extended with application-specific pointcut predicates. In essence, we combine the advantages of an extensible framework for defining advice expressions with the advantages of a dedicated and extensible pointcut language. In the remainder of this section, we introduce AspectS, CARMA, and their integration called AspectSOUL. In subsequent sections, we focus on how the open, logic-based pointcut language provides developers with an adequate means to handle complex and hard-to-maintain pointcut expressions.

2.1 AspectS

In the AspectS framework, aspects are implemented as subclasses of the class `AsAspect`. Its advices can be implemented as methods whose name begins with `advice` and which return an instance of `AsAdvice`. Two of the subclasses of `AsAdvice` can be used to implement either an around advice or a before/after advice. An instance can be created by calling a method which takes as its arguments qualifiers, a block implementing the pointcut, and blocks to implement the before, after or around effects of the advice.

An example advice method is shown in Figure 1. It specifies that any invocation of an `eventDoubleClick:` method implemented by `WindowSensor` or any of its subclasses should be logged. The effect of the advice is implemented in the block passed to the `beforeBlock:` parameter. When one of the methods specified by the pointcut needs to be executed, this block is executed right before the execution of the method's body. The block is passed a few arguments: the receiver object in which the method is executed, the arguments passed to the method, the aspect and the client. In this example, the block simply logs some of its arguments to the transcript. Note that it calls a method on `self`, aspect classes can implement regular methods besides advice methods as well. The pointcut is implemented by the block passed to the `pointcut:` argument. It returns a collection of `AsJoinpointDescriptor` instances. This collection is computed using the Smalltalk meta-object protocol and collection enumeration messages: the collection of `WindowSensor` and all of its subclasses is filtered to only those that implement a method named `eventDoubleClick:`, an `AsJoinpointDescriptor` is then collected for each of these.

Advice qualifiers specify dynamic conditions that should hold if the advice is to be executed. These conditions are implemented as activation blocks: blocks that take as arguments an aspect object and a stack frame. The framework defines a

```
adviceEventDoubleClick

 ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier attributes: #(receiverInstanceSpecific))
    pointcut: [
      WindowSensor withAllSubclasses
        select: [:each |
           each includesSelector: #eventDoubleClick:]
        thenCollect: [:each |
           AsJoinPointDescriptor targetClass: each targetSelector: #eventDoubleClick:]]
    beforeBlock: [:receiver :arguments :aspect :client |
      self showHeader: '>>> EventDoubleClick >>>'
           receiver: receiver
           event: arguments first]
```

Fig. 1. Example advice definition in AspectS

number of activation blocks, that fall in two categories: checks done on the top of the stack, or on lower levels of the stack. The former are used for example to restrict advice execution to sender/receiver-specific activation: an advice on a method is only executed if the method is executed in a specific receiver object, or was invoked by a specific sender object, or is associated with a specific thread of control. The latter are used for control-flow related restrictions, such as only executing an advice on a method if the same method is not currently on the stack. The activation blocks have names, which are specified in the attributes of an `AsAdviceQualifier`. In the example advice, one activator block is specified: `receiverInstanceSpecific`.

Aspects can be woven into the Smalltalk image by sending an explicit `install` message to an aspect instance. The `install` method collects all advice objects in the class and executes their pointcut blocks to get the collection of joinpoint descriptors. The methods designated by these descriptors are then decorated by wrappers [11], one for each advice affecting this particular method. The wrappers check the activation blocks specified in their advice, passing them the aspect and the top stack frame (accessed using the `thisContext` reflective feature of Smalltalk [12]). If an activation condition does not hold, the wrapper simply executes the next wrapper (if any), or the original method. If all activation conditions hold, the wrapper executes the advice's around, before, and/or after block, and then proceeds to the next wrapper (if any) in the proper order, or the original method.

2.2 CARMA

CARMA is a pointcut language based on logic meta programming for reasoning about dynamic joinpoints. Unlike pointcuts in AspectS, CARMA pointcuts do not express conditions on methods, its joinpoints are representations of dynamic events in the execution of a Smalltalk program. CARMA defines a number of logic predicates for expressing conditions on these joinpoints, and pointcuts are written as logic queries using these predicates. It is possible to express conditions on dynamic values associated with the joinpoints. Furthermore, logic predicates are provided for querying the static structure of the Smalltalk program. These predicates are taken from the LiCoR library of logic predicates for logic meta programming [13]. The underlying language of this library and CARMA is the SOUL logic language [13,10].

The SOUL logic language is akin to Prolog [14], but has a few differences. Some of these are just syntactical, such as that variables are notated with question marks rather than capital letters, the "`:-`" symbol is written as `if`, and lists are written between angular (`<>`) instead of square brackets (`[]`). More importantly, SOUL is in linguistic symbiosis with the underlying Smalltalk language, allowing Smalltalk objects to be bound to logic variables and the execution of Smalltalk expressions as part of the logic program [15]. The symbiosis mechanism is what allows CARMA to express conditions on dynamic values associated with joinpoints which are actual Smalltalk objects, such as the arguments of a message.

The advantage of building a pointcut language on the logic programming paradigm lies in the declarative nature of this paradigm. No explicit control structures for looping over a set of classes or methods are necessary in pointcuts, as this is hidden in the logic language [16]. A pointcut simply states the conditions that a joinpoint should meet in order to activate an advice, without specifying how those joinpoints are computed. This makes declarative pointcuts, given some basic knowledge of logic programming of course, easier to read. A logic language also provides some advanced features such as unification that make it easier to write advanced pointcuts. A full discussion is outside the scope of this paper, but a more comprehensive analysis was given in earlier work [9]. In the next sections, we will however show how some of these features – particularly the ability to write multiple rules for the same predicate – are useful for writing model-based pointcuts.

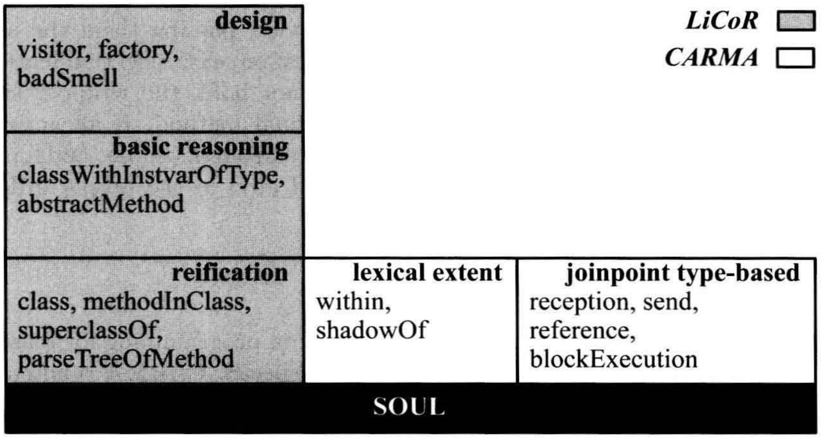
design
visitor, factory, badSmell
basic reasoning
classWithInstvarOfType, abstractMethod
reification
class, methodInClass, superclassOf, parseTreeOfMethod
lexical extent
within, shadowOf
joinpoint type-based
reception, send, reference, blockExecution
SOUL
LiCoR
CARMA

Fig. 2. Organization of, and example predicates in LiCoR and CARMA

The predicates in CARMA and LiCoR are organized into categories, as shown in Figure 2. The LiCoR predicates are organized hierarchically, with higher predicates defined in terms of the lower ones. The predicates in the "reification" category provide the fundamental access to the structure of a Smalltalk program: these predicates can be used to query the classes and methods in the program, and the fundamental relations between them such as which class is a superclass of which other class. The "basic reasoning" predicates define predicates that can be used to query more complex relations: which classes indirectly inherit from another class, which methods are abstract, which types an instance variable can possibly have etc. The "design" category contains predicates about design information in programs: there are for example predicates encoding design patterns [17] and refactoring "bad smells"[18].

The CARMA predicates access the dynamic structure of a Smalltalk program. There are two categories of predicates in CARMA, neither is defined in terms of each other, nor in terms of the LiCoR predicates. Nevertheless, the purpose of the "lexical extent" predicates is to link the dynamic and static structure, so that reasoning about both can be mixed in a pointcut. The `within` predicate for example can be used to express that a joinpoint is the result of executing an expression in a certain method. The "type-based" joinpoint predicates are the basic predicates of CARMA, they express conditions on certain types of joinpoints and basic data associated with those. An example is the `reception` predicate which is used to express that a joinpoint should be of the type "message reception", which means it represents the execution of a message to an object. Besides the joinpoint, the predicate has parameters for the basic associated data: the selector of the message and its arguments. There are also a few other predicates in CARMA (not shown in the figure), such as the `inObject` predicate which links a joinpoint to the object in which it is executed. In the case of a reception joinpoint, this is the receiver of the message.

A pointcut in CARMA is written as a logic query that results in joinpoints. By convention, the variable to which these are bound is called "`?jp`". The joinpoint representations should only be manipulated through the predicates provided by CARMA. An example pointcut is given in the next section.

2.3 CARMA Pointcuts in AspectS

AspectSOUL, the integration of CARMA with AspectS, is realized by subclassing the advice classes of AspectS so that a CARMA pointcut can be specified instead of a Smalltalk expression. The signature of the instance creation message for these subclasses is similar to the original. It takes as arguments a string with a CARMA pointcut, qualifiers and an around or before and/or after block. The message does a mapping to the instance creation message of the superclass. This is not a direct 1-on-1 mapping however, because CARMA pointcuts are about dynamic joinpoints, in contrast with the more static joinpoints of AspectS. Also, because AspectS does not support aspects that intercept block execution nor variable accesses or assignments, these features of CARMA are not adopted in AspectSOUL.

An example of an AspectS advice with a CARMA pointcut is shown in Figure 3. This is an around variant of the first example advice, with a pointcut that has the same effect. The first condition in the pointcut specifies that `?jp` must be a message reception joinpoint, where the selector of the message is `eventDoubleClick:`. The arguments of the message are bound to the variable `?args`. However, `?args` is not used any further in the pointcut which expresses that no conditions are put on the argument list. The second condition expresses that the joinpoint must occur lexically in a method with name `?selector` in the class `?class`. For a message reception joinpoint, this is effectively the method that is executed to handle the message. The final condition expresses that the class `?class` should be in the hierarchy of the class `WindowSensor`. The block has the same effect as in the first example, except that here it explicitly calls the next wrapper (if any) or original method.

```
adviceEventDoubleClick

 ^ AsCARMAAroundAdvice
    qualifier: (AsAdviceQualifier attributes: #())
    pointcutQuery: 'reception(?jp, #eventDoubleClick:, ?args),
                    within(?jp, ?class, ?selector),
                    classInHierarchyOf(?class, [WindowSensor])'
    aroundBlock: [:receiver :arguments :aspect :client :clientMethod |
      self showHeader: '>>> EventDoubleClick >>>'
           receiver: receiver
           event: arguments first.
      clientMethod valueWithReceiver: receiver arguments: arguments]
```

Fig. 3. Example AspectS advice definition with a CARMA pointcut

```
reception(?jp, #eventDoubleClick:, <?event>),
objectTestHolds(?event, #isYellow)
```

Fig. 4. A CARMA pointcut with a condition on a dynamic value

Figure 4 gives an example of a CARMA pointcut which does express conditions on the arguments of a message reception. The first condition expresses that `?jp` must be a message reception joinpoint of the message `eventDoubleClick:`, where the argument list unifies with the list `<?event>`. Thus the argument list has to have one argument, which is bound to the variable `?event`. The value of `?event` is the actual Smalltalk event object that is sent as the argument of `eventDoubleClick`. The second condition uses the `objectTestHolds` predicate, which uses the symbiosis mechanism of SOUL to express that the object in `?event` must respond `true` to the message `isYellow`. Thus, this pointcut captures joinpoints when a message about a double click event of the yellow mouse button is sent to some object. Expressing the same in AspectS can only be done by defining an appropriate qualifier, or by including the dynamic condition in the around block of the advice. The CARMA approach means that what conceptually should go into a pointcut can be better separated from the effect of the advice: that we only want to intercept double click events of the yellow mouse button is part of the "when" of the advice, not of the "what effect" it has. All of the qualifiers of AspectS can be expressed in CARMA, except for the control-flow qualifiers because CARMA does not currently support a construct similar to the `cflow` pointcut of AspectJ [19].

Two-phased weaving: The mapping done in the AspectSOUL advice subclasses to the original advice classes of AspectS involves the two-phase weaving model of CARMA. Because CARMA allows dynamic conditions and it is a Turing-complete language, it requires some advanced techniques to optimize weaving [9]. The mapping uses abstract interpretation [20] of the pointcuts to determine the methods which *may* lead to joinpoints matching the pointcut. For the pointcut of Figure 4, it determines that only executions of methods named `eventDoubleClick:` may match the pointcut. For these methods, `AsJoinpointDescriptors` are generated and passed to the advice superclass. The effect block passed to the superclass is wrapped so that it at run-time