The JCop Language Specification

Malte Appeltauer, Robert Hirschfeld

Technische Berichte Nr. 59

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Universität Potsdam

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 59

Malte Appeltauer | Robert Hirschfeld

The JCop Language Specification

Version 1.0, April 2012

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2012
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2012/6020/
URN urn:nbn:de:kobv:517-opus-60208
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-60208

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-193-6

Contents

Abstract

Program behavior that relies on contextual information, such as physical location or network accessibility, is common in today's applications, yet its representation is not sufficiently supported by programming languages. With *context-oriented programming* (COP), such context-dependent behavioral variations can be explicitly *modularized* and *dynamically activated.* In general, COP could be used to manage any context-specific behavior. However, its contemporary realizations limit the control of dynamic adaptation. This, in turn, limits the interaction of COP's adaptation mechanisms with widely used architectures, such as event-based, mobile, and distributed programming.

The *JCop* programming language extends Java with language constructs for context-oriented programming and additionally provides a domain-specific aspect language for declarative control over runtime adaptations. As a result, these redesigned implementations are more concise and better modularized than their counterparts using plain COP.

JCop's main features have been described in our previous publications. However, a complete language specification has not been presented so far. This report presents the entire JCop language including the syntax and semantics of its new language constructs.

1 Introduction

In most object-oriented programming languages, any method execution is influenced by its execution context, for example the values of its arguments or the state of referenced objects. For a single method, such behavioral variations can be well modularized by the common means of object-oriented languages, such as method parameterization and overriding. Programs may additionally use conditional statements and inheritance to separate the variation code from the default behavior.

Context-oriented programming [11, 6] (COP) addresses the representation of behavioral variations that impact several method executions simultaneously. Their implementation often crosscuts the class-based decomposition. With common object-oriented means, these kinds of *crosscutting behavioral variations* cannot be sufficiently modularized [8][1]. In addition, context information must be propagated within a control flow and exposed to the respective methods. This requires addi-

[1] In the following, we consider COP as a extension of object-oriented programming, though its basic concepts can be applied to other paradigms as well. In fact, COP has been originally

tional infrastructural code for both the context propagation and the dispatch to the required variation. In the following, we refer to source code that implements behavioral variations as an *adaptation*. The code that triggers the execution of an adaptation is called a *composition* because it composes the base behavior with adaptations.

In COP, behavioral variations can be represented by special methods called *partial method declarations* that implement variations of particular methods. Partial method declarations are encapsulated by *layers*. Depending on this contextual information, adaptations may be explicitly composed for a control flow. Adaptation composition—in COP also called as *layer composition*—is expressed by special layer activation and deactivation statements. Layer activation is scoped to specific control flows. This prevents layer adaptations from unintentionally affecting other computations, which could lead to state inconsistencies and corrupted results.

JCop [1, 3, 16, 15] is our Java language extension that provides dedicated language constructs for COP, such as *layer* and *partial method* declarations, and *layer composition* statements. In addition, JCop offers linguistic means to overcome the issue of *scattered layer composition statements* that has been identified and described in previous work [3, 2]. Therefore, JCop integrates COP with a domain-specific aspect-oriented programming [12] (AOP) language.

Although JCop has been presented in previous publications, a complete language specification has not been presented, yet. This report presents the entire JCop language including the syntax and semantics of its new language constructs.

Section 2 gives an overview of the fundamentals of context-oriented programming that are necessary to understand JCop's features. Section 3 presents JCop's declaration constructs that have been added to Java: layer type declarations and partial method declarations. Section 4 discusses the control mechanisms for dynamic adaptation. Section 5 describes the context class declaration.

We present the syntax production rules in the following sections using the grammar notion of the Java Language Specification, third edition [9]. Terminals are shown in `fixed font`, non-terminals are shown in *italic*. Each line of a grammar definition represents one alternative right-hand side. The subscripted suffix '$_{opt}$' after elements denotes an optional symbol. The words '*one of*' after a grammar definition header starts an abbreviation that declares each of the following terminal symbols as alternative definitions. Some JCop production rules extend existing Java production rules, which we mark with underlines and abbreviate their definition using '...'.

implemented for Lisp [6].

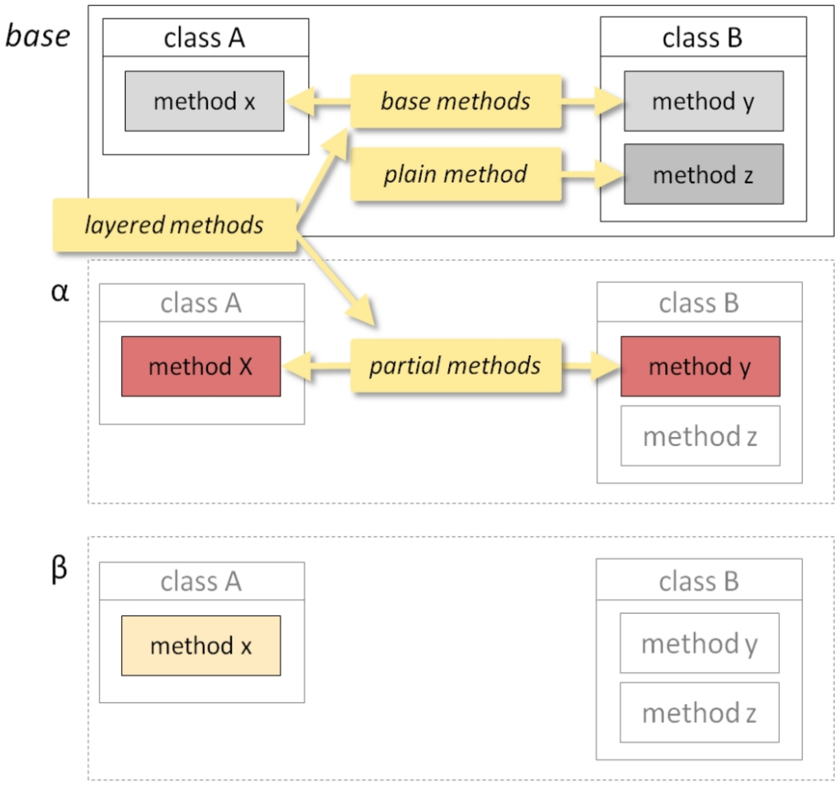


Figure 1: Modularization of behavioral variations by layers and partial methods.

2 Overview of Context-oriented Programming

In this section, we give an overview of the key features of COP. Subsection 2.1 introduces layer-based modularization. Subsection 2.2 describes the run-time composition of layers.

2.1 Modularization

Layers are modules that are conceptually orthogonal to classes, i.e., a layer may extend or replace the functionality of one or more classes. To distinguish between different kinds of method definitions, we use the following terms.

Plain method declaration denotes a common (Java) method that is not affected by layer adaptations.

Layered method declarations are the counterpart to plain methods. The term describes methods that are adapted by layers. A layered method declaration consists of a *base method* and at least one *partial method*.

Base method declarations are *plain methods* that are executed if no active layer provides a corresponding partial method. In most COP languages, the methods provided by the base language belong to an implicit *base layer*, which is why we use the term base method.

Partial method declarations are declared within a layer and implement a behavior variation of a base method.

Layer local methods are declared within a layer. They are only accessible from within the layer and can be referred by partial methods for a better modularization.

The enclosing class of a base method is called the *host class* of the base method and its partial method. Figure 1 illustrates the different kinds of method definitions.

2.2 Runtime Adaptation

At run-time, layers can be activated and deactivated. On layer activation, its partial methods are composed with their base methods and the partial methods of the other active layers. Therefore, layer activation and deactivation is called *layer composition*.

Layer composition influences the method lookup of a layered method. Listing 2 illustrates the activation of a layer `Alpha`. On activation, the layer and its methods are composed with the base system. During this composition, a call to `B.y` is first dispatched to the partial method of `Alpha`. Hence, the *layer-aware method dispatch* adds another dimension to the object-oriented method lookup. In addition to a message's receiving object and signature, the layer-aware method lookup also considers the message's contextual information that is implicitly passed through the control flow. A layer composition is expressed by a block statement and either implemented by a method or a special language construct. Layer composition is scoped to the *dynamic extend* of a block of statements[2].

In Figure 2, the method call `B.y` is first dispatched to the partial method of `Alpha`. To explicitly invoke the base method of `B.y` within the partial method, we can use

[2] This denotation is borrowed from Lisp [14]. A layer composition established at the start of the execution the composition block and disappears when that block finishes executing: the lifetime of the composition is tied to composition block

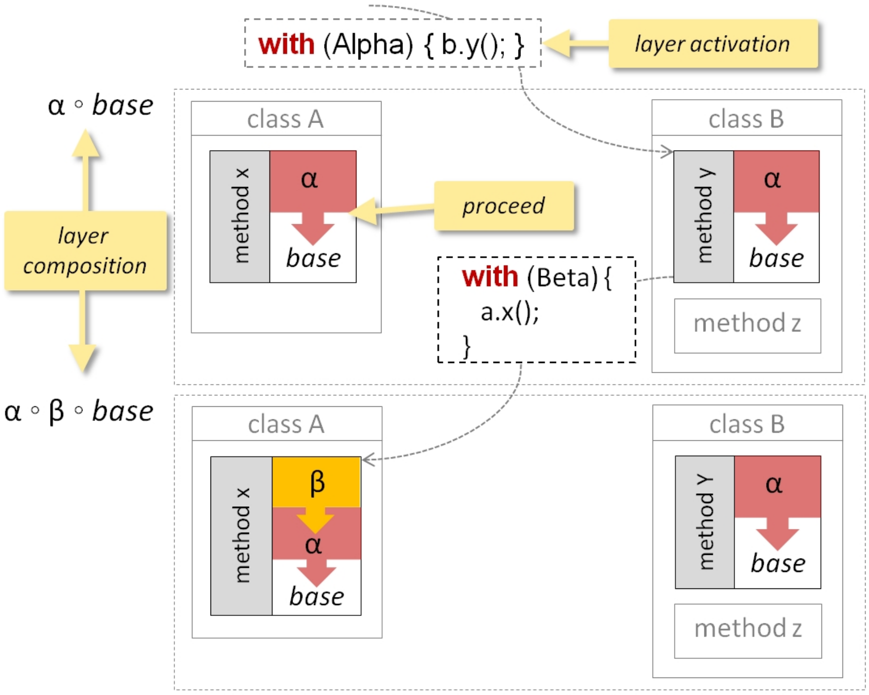


Figure 2: Dynamic composition of layers is controlled by layer activation.

`proceed`, a pseudo-method that allows invoking the next partial method definition (or the base method). Both the return type and the expected arguments of `proceed` must conform to the method's signature.

The base method of `B.y` then activates another layer, `Beta`, and calls the method `A.x`. Because this layer composition is executed within the dynamic extend of our previous layer composition, the method `A.x` consists of two partial methods (of `Beta` and `Alpha`) and the base method. Partial method definitions can be declared to be executed before, after, around, or instead of the base method definition. The layer `Beta` of our example declares its partial method of `A.x` to be executed after the base method. Therefore, the method call is first dispatched to `Alpha`, then to the `A`, and finally to `Beta`. Layers can also be explicitly deactivated for a specific control flow by a similar construct.

3 Layer Modularization

This section presents JCop's language constructs for the modularization of adaptations. Subsections 3.1 and 3.2 describe the two main constructs, layers and partial methods. Subsection 3.3 presents layer inheritance mechanisms.

3.1 Layer Declarations

JCop extends the Java type system with *layer type declarations* that modularize behavioral variations. Layer declarations can contain *partial method declarations* in addition to the standard member declarations. This section describes the properties of layer declarations that implement *class-in-layer* adaptations. It also explains open layer declarations that allow the extension of a layer declaration with additional partial method declarations, which is called *layer-in-class* adaptation [11].

3.1.1 Layer Type Declaration

Like a top-level class, a layer is declared in its own compilation unit and specifies a new, named reference type. Unlike classes, nested and anonymous layer declarations are not allowed. However, declarations can be opened in any class (or nested class[3]) to add partial member declarations, see Subsection 3.1.2.

```
ClassDeclaration :
   ...
   LayerDeclaration
   ContextClassDeclaration

LayerDeclaration :
   LayerModifiers_opt layer Identifier Super_opt Interfaces_opt
       LayerBody
```

Layer declarations use the keyword `layer` instead of `class`. The keyword is optionally preceded by *modifiers* and followed by *superlayer* and *interface* declarations. A layer declaration implicitly inherits from `jcop.lang.Layer`. The `extends` clause known from class declarations can be used to specialize the super type. Layers can only inherit from other layers; they cannot inherit from classes. Layers can implement interfaces in the same way classes do. Layers extend the inheritance mechanism for partial members, see Subsection 3.3.

Listing 1 presents an example consisting of two classes that are adapted by two layers. The listing is an implementation of the layers depicted in Figures 1 and 2. The layer `Alpha` adapts methods of both classes—thus, it implements crosscutting behavior. The layer `Beta` adapts the method `B.z` that is also adapted by `Alpha`. If both layers are composed together, their composition order determines which is executed first.

[3] The Java Language Specification defines a nested class as a general term for *static nested classes* and *inner classes* (*non-static nested classes*) [9].

```
public class A {
  public int x() { ... }
}

public class B {
  public int y() { ... }
  public int z() { ... }
}
public layer Alpha {
  public int A.x() { ... }
  public int B.y() { ... }
}

public layer Beta {
  public int B.x() { ... }
}
```

Listing 1: Two layers adapting the methods of two classes.

3.1.2 Open Layer Declaration

The implementation of behavioral variations can be regarded from two perspectives. A developer may either focus on the *commonality* of the partial methods that implement a behavioral variation (i.e., their interaction among themselves), or on their *individuality* (i.e., their interaction with its host class). If the focus is on commonality, the partial methods should be implemented within a top-level layer declaration. In this case, source code changes that affect several partial methods are specified locally in the top-level layer declaration. However, if the focus is on individuality, the partial methods should be implemented close to their base method definition rather than in their top-level layer declaration.

We implement the second approach by using an *open layer* technique, similar to the *open class* concept [5, 17]. As a result, JCop layer definitions can be opened in classes and extended with additional partial methods. Syntactically, open layer declarations are class body declarations. They can only contain partial methods of base methods that are defined or inherited by the enclosing class. This mechanism also allows adaptation of methods with restricted visibility, see Subsection 3.2.2. Listing 2 shows a declaration of a partial method for the private method `A.x` within an open layer declaration in `A`.

```
ClassBodyDeclaration :
  ...
  OpenLayerDeclaration

OpenLayerDeclaration :
  layer Identifier { OpenLayerBodyDeclaration_opt }

OpenLayerBodyDeclaration :
  ClassMemberDeclaration
  LocalPartialMethodDeclaration
```

```
public class A {
  private void x() { ... }
  // open layer declaration
  layer Alpha {
    private void x() { ... }
  }
}
```

```
public class B {
  public int y() { ... }
}

public layer Alpha {
  public int B.y() { ... }
}
```

Listing 2: Top-level and open layer declaration of layer `Alpha`.

```
import lib.*;
public class A {
 public int x() {
   return new LibClass().aMethod();
 }
}

public staticactive layer Alpha {
  public int lib.LibClass.aMethod() { ... }
}
```

Listing 3: A layer declaration using the `staticactive` modifier.

3.1.3 Layer Modifiers

Layer access can be declared by the default Java modifier, `public`, `protected`, and `private`. These modifiers have the same effect on layers as on classes. In addition, layers can be declared `abstract`, which means concrete subtypes must then implement all abstract layer members.

By default, layers are composed per control-flow. In addition, the modifier `staticactive` declares that one *singleton* instance of the layer is implicitly globally activated on static initialization of the layer declaration. For the initialization of the singleton, the default constructor of the layer is used. This feature simplifies, for example, the adaptation of library methods that need to be active for the whole application life cycle, as shown in Listing 3.

```
LayerModifiers :
  LayerModifier
  LayerModifiers LayerModifier

LayerModifier : one of
  ClassModifier staticactive
```

3.2 Partial Method Declarations

A layer body declaration may contain partial method declarations, and class member declarations (fields, inner classes, interfaces, and methods). The execution order of partial methods is controlled using special *adaptation modifiers* . Partial method declarations qualify the base method to be adapted by their *signature*. They are executed in the scope of the object to be adapted but declared within layers. The scope of both the object and layer can be accessed through special keywords.

```
LayerBody :
  { LayerBodyDeclarations_opt }

LayerBodyDeclarations :
  LayerBodyDeclaration
  LayerBodyDeclarations LayerBodyDeclaration

LayerBodyDeclaration :
  ClassMembersDeclaration
  LayerMemberDeclaration

LayerMemberDeclaration :
  PartialMethodDeclaration
```

3.2.1 Partial Method Modifiers

Partial methods can declare additional modifiers, the *partial method modifiers*. They only affect partial method declaration, they do not affect the corresponding base method. The set of partial method modifiers consists of `final`, `abstract`, and the *adaptation modifiers* `before` and `after`.

```
PartialMemberModifiers :
  PartialMemberModifier_opt Modifiers

PartialMemberModifier : one of
  before after final abstract
```

Adaptation modifiers can declare the execution order of partial methods and the base method in the following way:

Execute before the base method If the partial method's signature contains the `before` modifier, the variation will be executed before the original join point. In that case, partial methods are void and do not contain a return statement.

```
1 package p;
2
3 public class A {
4   public int x(int i) { ... }
5 }
6 public layer Alpha {
7   public int p.A.x(int i) { ... }
8   before public int p.A.x(int value) { ... }
9   after public int p.A.x(int value) { ... }
10 }
```

Listing 4: Partial method declarations using adaptation modifiers.

Execute after the base method A partial method containing an `after` modifier will execute its method block after the base method. This execution time time corresponds to a the execution of a `finally` block after a return statement of a plain Java method.

Execute around the base method The pseudo-method call `proceed` is used to proceed to other method variations, see Section 4.1.3. It can be used in any partial method to call the next partial method of the current composition chain. The expression can also be used multiple times in a partial method, causing multiple invocations of the next partial methods.

Execute instead the base method If no partial member modifier is declared, partial methods are executed *instead* of the base methods. Their method blocks are executed without calling other partial methods.

Layers can form a inheritance hierarchy and override partial method declarations, see Subsection 3.3. To force sublayers to implement a partial method, the method can be declared `abstract`. To prohibit sublayers from overriding a partial method, the method can be declared `final`.

3.2.2 Partial Method Signature

Partial method definitions are qualified by a *full qualified signature* that includes the base method's enclosing type name and the full package name. The latter guarantees a unique specification of the corresponding base method. At run-time, partial methods may replace their base method. Therefore, they must adhere to the base method's parameter and return types as well as to its modifiers. The modifiers may be preceded by an additional adaptation modifier but cannot be further refined. A layer may contain several partial methods for the same method that are preceded by a different adaptation modifier. For example, the partial method declaration of the method `A.x` in Listing 4 is extended by a full qualified identifier and can contain an adaptation modifier.

The first partial method of layer `Alpha` replaces its base method (Line 7). If `proceed` is called in its method body, the partial method is executed around its base method,

modifier	**adaptability**
`private`, `protected`	open layer declarations only
default	layers of same package and open layer declarations
`public`	any layer declaration
`final`, `abstract`, `native`	no adaptation
`static`, `volitaire`, `strictfp`	no direct effect

Figure 3: Effect of method modifiers to the adaptability by partial methods.

see Section 4.1.3. The second partial method (Line 8) is executed before the base method (Line 4); the third partial method (Line 9) after the base method. Methods with `before` or `after` modifier must not contain a return statement (regardless of their return type), but can use `proceed`.

A method's modifiers also apply to its partial methods. Hence, the partial methods must declare the same modifiers. The effect of these modifiers must be respected by layer adaptations; modifiers may restrict or even prohibit adaptation by a partial method, as shown in Figure 3: In general, partial methods can be declared for any method visible to the respective layer. The partial method declarations of top-level layer declarations can adapt any method that is accessible—with respect to its encapsulation rules—by the layer. To adapt a method with default visibility, the layer must be declared in the same package as the method's class. Top-level layers cannot declare partial methods for private or protected methods. Private and protected methods can only be adapted by *local partial methods*.

A method that is declared as *final* cannot be adapted by a layer just as it cannot be overridden by subclasses. Also, abstract methods cannot be adapted since they cannot be executed. However, their overridden methods in subclasses can be adapted. JCop does not support adaptation of native methods. Any incorrect declaration of partial methods will cause a compile-time error. Note that because of the aforementioned restrictions, the use of the modifiers `abstract` and `final` with partial methods is unambiguous: The modifiers cannot be part of the signature of the base method but instead refer to the partial method.

```
PartialMethodDeclaration :
 PartialMethodHeader MethodBody

PartialMethodHeader :
   PartialMemberModifiers_opt ResultType LayerMethodDeclarator
                                                    Throws_opt

LayerMethodDeclarator :
 FullQualifiedName ( FormalParameterList_opt )
FullQualifiedName :
  TypeName . Identifier
```

A local partial method is declared in an open layer declaration, see Subsection 3.1.2. Local partial methods can only adapt methods of the enclosing class declaration (or its super types). They do not use a full qualified signature. Instead, they use the same as their plain methods, optionally preceded by an adaptation modifier.

```
LocalPartialMethodDeclaration :
 LocalPartialMethodHeader MethodBody

LocalPartialMethodHeader :
   PartialMemberModifiers_opt ResultType MethodDeclarator
                                                      Throws_opt
```

3.2.3 Partial Method Body Scoping

The scope of a class member declaration in, or inherited by, a layer type `Alpha` is the entire body of `Alpha`, including any nested type declarations. The scope of layer member declarations in a layer type `Alpha` is the target class which is to be adapted. As a consequence, the behavior of Java's keywords `this` and `super` must be specified for their use within partial method definitions. The two methods of the layer `Alpha` in Listing 5 give concrete examples of the scoping. Member references without an explicit receiver object or `this` keyword are first looked up in the enclosing layer, then in the scope of the target object (Line 22). In partial methods, the `this` keyword refers to the object to be adapted (Line 23) and the `super` keyword does not refer to the layer's super type but to the super type of the partial method's target object (Line 25). Therefore, for the explicit access of the enclosing layer within partial methods, we introduce two new keywords: `thislayer` refers to the enclosing layer and `superlayer` refers to the super type of the enclosing layer. Without declaring any receiver object, the invocation is dispatched to the layer-local method, which is equivalent to `thislayer.m1`. Using `this`, the method call is passed to `A` or to partial methods of `m1`.

```
Access :
  ...
  thislayer
  superlayer
```

```
public class SuperA {
  public void n() {
    ...
  }
}
public class A extends SuperA {
  public void m() {
    ...
  }
  public void n() {
    ...
  }
}
public layer SuperAlpha {
  protected void n() {
    ...
  }
}
```

```
public layer Alpha extends SuperAlpha {
  // executed in the scope of A
  public void A.m(){
    n();              // Alpha.n()
    this.n();         // A.m1()
    thislayer.n();  // Alpha.n()
    super.n();        // SuperA.n()
    superlayer.n(); // SuperAlpha.n()
  }
  // executed in the scope of Alpha
  private void n() {
    n();              // Alpha.n()
    this.n();         // Alpha.n()
    // 'thislayer' is undefined
    super.n();        // SuperAlpha.n()
    // 'superlayer' is undefined
  }
}
```

Listing 5: Example for scoping of layer method bodies.

3.3 Layer Inheritance

The implicit super type of a layer is `jcop.lang.Layer`. Optionally, layers can declare a super layer using Java's `extends` declaration. Layers may only inherit from other layers. They cannot inherit from classes. The scope of the super layer can be accessed by the `superlayer` keyword.

A partial method m_{Alpha} declared in a layer `Alpha` overrides a partial method, $m_{SuperAlpha}$, declared in layer `SuperAlpha` if all of the following are true:

1. `Alpha` is a sublayer of `SuperAlpha`.

2. The signature of m_{Alpha} is *equal* to the signature of $m_{SuperAlpha}$, including all partial method modifiers (except `abstract`[4]).

Listing 6 demonstrates how abstract layers and partial methods can be used to define adaptation points that can be implemented by concrete layers. The example shows an abstract layer that adapts methods that are relevant for file creation and access. Concrete layers can implement different logging styles or replace the original file interaction by database access.

Layers can also inherit from concrete layers and override concrete partial methods. Partial methods do not appear in the interface of layers (since their scope is the class to be layered). Thus the implementation of a partial method in a super layer cannot be invoked by a super call like `super.partialMethod`[5]. To support super

[4] `SuperAlpha` and $m_{SuperAlpha}$ may be declared abstract and implemented by m_{Alpha}.

[5] Otherwise, the same mechanism would allow explicit calls of any partial method of a super

```
abstract public layer FileAccess {
  abstract  public File FileHandler.create(String name);
  abstract  public String FileReader.read(File toBeRead);
}

public layer FileAccessLogging extends FileAccess {
  public File FileHandler.create(String name) {
    System.out.println("create file");
    return proceed(name);
  }
  public String FileReader.read(File toBeRead) {
    System.out.println("read file");
    return proceed(toBeRead);
  }
}
```

Listing 6: Layer hierachy with abstact partial methods.

access for partial methods, JCop provides a variation of the `proceed` expression, `superproceed`, that allows access to a partial method's super declaration. The use of `superproceed` is statically checked. Therefore, if no super layer implements a corresponding partial declaration its use causes a compile-time error. The layer `VerboseFileAccessLogging` in Listing 7 extends `FileAccessLogging` of the previous example and attaches additional information to the logging. The original logging functionality is invoked by `superproceed`.

4 Layer Adaptation and Composition

Behavioral variations of layers are composed at run-time and scoped to the execution of one or more expressions. If active, a partial method superimposes its base method and receives all method invocations first. These semantics are referred as *sideways composition* [11]. COP languages define layer composition by an *explicit statement* that wraps the expressions to be adapted.

In this section, we present the semantics of layer composition in JCop and JCop's means for layer composition. Subsection 4.1 explains the semantics of layer activation and deactivation, the adaptation modifiers, and the `proceed` instruction. Subsection 4.2 describes the language constructs for explicit layer composition; Subsection 4.3 introduces reflective access to layers and layer composition, and demonstrates reflective layer activation.

JCop additionally supports a *declarative specification* of adaptation points in a dedicated class, the so called *contextclass*. A full discussion of this aspect of JCop

layer, with unclear semantics.

```
public layer VerboseFileAccessLogging extends FileAccessLogging {
  public File FileHandler.create(String name) {
    File createdFile = superproceed(name);
    System.out.println("file name: " + name);
    return createdFile;
  }
  public String FileReader.read(File file) {
    String content = superproceed(file);
    String log = String.format("file name: %s content: %s",
                               file.getName(), content);
    System.out.println(log);
    return content;
  }
}
```

Listing 7: Usage of `superproceed` in partial methods

layer composition is provided in Section 5.

4.1 Semantics of Layer Composition

Layer composition can be expressed by the two functions *with* and *without* that operate on sets of layers. These two functions allow dynamic activation or exclusion of layers at any point in the control flow. To specify these control flow locations, JCop provides different composition constructs.

In the following, we formulate the semantics of layer composition. Let *base* be the base layer and L the set of layers[6] l_i in a program:

$$L = \{l_1, ..., l_i, ..., l_n\}, 1 \leq i \leq n \in \mathbb{N}.$$

An ordered list of layers is then denoted as tuple $(l_1, ..., l_n)$. We define two layer tuple sets, $\overline{L}$ and C. $\overline{L}$ represents the set of tuples that may be activated by a composition. These tuples must not contain the base layer.

$$\overline{L} = \{(l_1, ..., l_i, ..., l_{n_k}, base) \mid l_i \in L \wedge 1 \leq i \leq n_k \in \mathbb{N}\}$$

A layer composition list is a layer tuple whose last element is the base layer. C describes the set of all possible layer compositions:

[6] Layers in JCop are instantiable. However, for layer composition, we can omit that implementation detail. Thus, an element l_i refers to a layer instance. It is possible that two tuple elements l_x and l_y refer to the same layer instance.

$$C = \overline{L} \times \{base\}$$

The following subsections present the semantics of the composition functions (Subsection 4.1.1) and the effect of adaptation modifiers (Subsection 4.1.2) and `proceed` to the layer dispatch (Subsection 4.1.3).

4.1.1 Composition Functions

The active layer composition can be manipulated by the two composition functions, *with* and *without*. As a first step, we omit the presence of adaptation modifiers and assume that layers, e.g, their partial methods, always proceed to the next layer in the composition.

The *with* function defines the join of a composition tuple $\in C$ with a tuple $\in \overline{L}$. This operation is called *layer activation*:

$$\begin{array}{llcccl} \mathit{with}: & C & \times & \overline{L} & \rightarrow & C \\ & (c_1, ..., c_n, base) & \times & (l_1, ..., l_m) & \mapsto & (l_m, ..., l_1, c_1, ..., c_n, base) \end{array}$$

Note that the layers $(l_1, ..., l_m)$ of the right-hand side of *with* are attached to the composition in reverse order.

The following examples use infix notation and demonstrate the effect of the function to the composition. The left-hand side describes the initial composition which will be joined to the right hand side tuple. The first example activates a layer *alpha* while no other layer (except for the base layer) is active. In this case, the call of a method layered by *alpha* would be dispatched first to the layer, and then to the base method.

$$(base) \ \mathit{with} \ (alpha) \mapsto (alpha, base)$$

In the following example, we append a layer to the head of the active composition. Thus, the last activated layer is accessed first.

$$(alpha, base) \ \mathit{with} \ (beta) \mapsto (beta, alpha, base)$$

JCop does not prohibit composing the same layer multiple times:

$$(beta, alpha, base)\ \ \mathit{with}\ \ (alpha) \mapsto (alpha, beta, alpha, base)$$

The *without* function removes—or *deactivates*—the tuple elements of its second argument from the composition. With other words, the result tuple contains the maximal number of layers of the first tuple that are not contained in the second tuple while preserving the original order:

$$\begin{array}{lcccc}
\mathit{without}: & & & & \\
C & \times & \overline{L} & \rightarrow & C \\
(c_1, ..., c_n, base) & \times & (l_1, ..., l_m) & \mapsto & (c_{i_1}, ..., c_{i_k}, base) \\
 & & & & c_{i_j} \in \{c_1, ..., c_n\} \backslash \{l_1, ..., l_m\}, \\
 & & & & 1 \leq j \leq k\ \wedge\ i_r < i_s\ for\ r < s \wedge \\
 & & & & k\ beeing\ maximal
\end{array}$$

The deactivation of a layer that is not part of the composition has no effect.

$$(alpha, base)\ \ \mathit{without}\ \ (beta) \mapsto (alpha, base)$$

If a *without* operation only affects layers at the head of a composition list, the respective layers can be simply removed.

$$(beta, alpha, base)\ \ \mathit{without}\ \ (beta) \mapsto (alpha, base)$$

If layers are removed at another position in the composition list, the list is reordered accordingly. A layer deactivation removes any occurrence of this layer from the composition list.

$$(alpha, beta, alpha, base)\ \ \mathit{without}\ \ (alpha) \mapsto (beta, base)$$

Any operation on the layer composition list only affects the dynamic extend of the current execution. Once the control flow returns from that execution, the composition is set to its old value.

4.1.2 Effect of Adaptation Modifiers

The composition functions *with* and *without* define the general order in which partial methods will be executed. In addition, partial methods themselves can influence their location in the dispatch chain by their adaptation modifiers.

For illustration, we consider the dispatch of a base method `C.x` and its partial methods. We assume that each layer in $(l_1, ..., l_n)$ provides one partial method for `C.x`[7] . The layer l_i^{mod} denotes a method's (for example, `C.x`) partial method that is declared in layer l_i and uses the adaptation modifier $mod \in Mod$. The adaptation modifiers are defined by the set:

$$Mod = \{before, after, around, base\}.$$

Its elements $before$ and $after$ represent the adaptation modifiers explained in Subsection 3.2.1. The effect of *around* and *instead* methods on the composition order is the same. Thus, we represent both types by *around*. For convenience, the modifier *base* denotes the base method (hence, $l_i^{base} \equiv base^{base} \equiv base$).

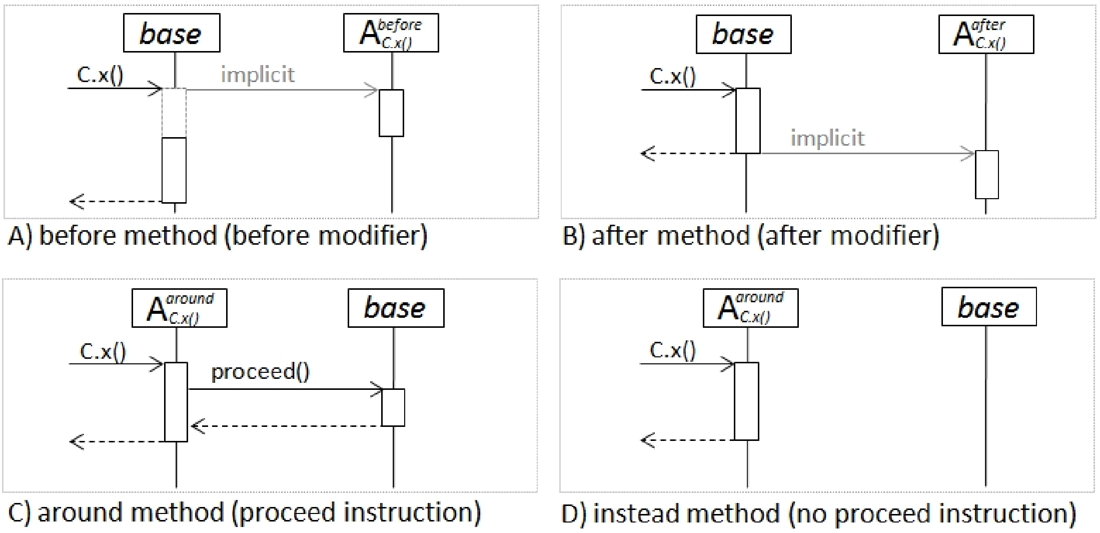
$$\begin{aligned} mod \ <_{adapt} \ mod' :\leftrightarrow \quad & (mod = before) \lor \\ & (mod = around \land mod'! = before) \lor \\ & (mod = base \land mod' = after) \lor \\ & (mod' = after) \end{aligned}$$

Based on this partial order, we define the dispatch order of partial methods. The dispatch order of two active partial methods is then defined as follows.

$$l_i^{mod} \quad <_{pmo} \quad l_j^{mod'} :\leftrightarrow \quad (mod \ <_{adapt} \ mod') \lor \quad (mod = mod' \land i \leq j)$$

The function *pmo* orders a tuple according to $<_{pmo}$. It is applied right before the dispatch of a layered method (i.e., right before the execution of `C.x`). The following two examples demonstrate the effect of the reordering to the dispatch chain. Note that the reordering does not affect the layer composition order but only the execution

[7] A layer could provide up to three partial methods for x that possess different adaptation modifiers. However, for our formulation, we can ignore that fact and represent a layer l with three partial methods as tuple $(l^{before}, l^{around}, l^{after})$.



A) before method (before modifier)

B) after method (after modifier)

C) around method (proceed instruction)

D) instead method (no proceed instruction)

Figure 4: Execution order of *before* (A), *after* (B), *around* (C), and *instead* methods (D).

chain for the respective layered method. Reordering is only necessary if the partial method contains adaptation modifiers other than *around* (and *base*). If not, *pmo* has no effect to the execution order.

$$pmo((beta^{around}, alpha^{around}, base)) = (beta^{around}, alpha^{around}, base) \tag{1}$$

If $before$ and $after$ are used, all *before* methods are moved to the compositions head and the *after* methods to the tail, while preserving the original order between the set of *after* and *before* methods.

$$\begin{aligned} pmo((delta^{after}, alpha^{before}, beta^{after}, gamma^{before}, base)) = \\ (alpha^{before}, gamma^{before}, base, delta^{after}, beta^{after}) \end{aligned}$$

4.1.3 Proceeding Partial Methods

The previous sections presented the composition of layers to generate an ordered layer tuple. In the simplest case, we employ only *around* methods (as shown in Equation (1)) and a method call is only dispatched to the first partial method. This

```
class A {
  public int x(int i) {
   ...
  }
}

layer Alpha {
  public int A.x(int i) {
    return proceed(i * 2);
  }
}
```

Listing 8: A partial method that is executed around its base method.

```
class A {
  public int x(int i) {
    ...
  }
  public float y(int i, boolean b) {
    ...
  }
}

layer Alpha {
  public int A.x(int i) {
    return (int) y(i, true);
  }
}
```

Listing 9: A partial method that is executed instead of its base method.

method eventually returns to the call side without calling the next partial method (or the base method), much like a call to an overwriting method that does not automatically execute its super method. Proceeding through the layer composition can be declared either *implicitly* or *explicitly.*

So far, we introduced implicit execution of more than one partial method by adaptation modifiers. That means, a composition $(l_1^{before}, ..., l_m^{around}, ..., base, ..., l_n^{after})$ would only execute the *before* methods l_1 up to the first *around* method l_m and the *after* methods l_n. Figure 4 (top) presents the execution time of *before* and *after* methods. They can only execute side-effects; they can neither manipulate the underlying method's execution nor return any value. A *before* method is implicitly called before the execution of its underlying method; an *after* method is executed right after its underlying method returns from its computation.

Explicit proceeding from one partial method to the next can be declared using the pseudo-method call `proceed`. Figure 4 (bottom) shows an *around* method that uses `proceed` to access the base method and an *instead* method that replaces the base method by not using `proceed`. The `proceed` call may be used only in the body of a partial method; if it appears anywhere else, a compile-time error occurs because it expects an argument list to be passed to the next partial method. In this way, partial methods can also influence the execution of their base method by modifying the base method's arguments. Listing 8 shows an example of an *around* method that doubles the value of the original argument by using the `proceed` call. Listing 9 illustrates an *instead* method that executes another method instead of its base method.

4.2 Explicit Layer Composition

Explicit composition statements are defined in the source code around the expressions to be adapted. JCop provides two block statements for composition, `with` and `without`. Both can be used in any statement block (method bodies and initializers). They consist of a keyword, an argument list specifying the right-hand side of the adaptation function, and a block that contains the expressions to be executed with the composition. The argument list consists of layer type expressions denoting the layers to be activated. More precisely, expressions of type `Layer`[8], `Iterable<Layer>`, or `Layer[]` are valid arguments. However, the use of any other expression type will cause a compile time exception. If all `with` arguments are evaluated to an empty list (or `null`), no layer is activated. The specified composition is only active for the *dynamic extent* of the composition statement block, i.e. for the control flow of any expression within the block. This implies that the composition of a particular layer is confined to the threads in which the layer was explicitly composed. Layer composition does not propagate to new threads; they start with no layers being active. Subsection 4.3.1 describes how layer composition could be transfered to other threads using meta programming.

Explicit layer activation is implemented by the `with` statement. The first activation in Listing 10 is parameterized with an instance of `Alpha` (Lines 6–8). The layer activation scope is defined by the following block, which contains one statement. Activation statements can contain a list of layers or expressions of type `Layer` to be activated. The second composition shows the activation of `Alpha` and `Beta`, which is the return value of `expr` (Lines 10–12). Alternatively, `with` statements can be nested to activate multiple layers (Lines 14–18). The `without` statement implements the *without* function, which excludes specific layers from the current composition tupel. A deactivation removes all references to the deactivated layer in the current composition (Lines 22–24). Layer instances are considered as different layers. Therefore, a deactivation of one instance does not remove any other instances of the same layer. For convenience, JCop offers the `withoutall` construct to remove all instances (Lines 25–27).

```
StatementWithoutTrailingSubstatement :
   ...
   LayerCompositionStmt

LayerCompositionStmt       :
   LayerCompositionHeader Block

LayerCompositionHeader :
   LayerCompositionFunction( ArgumentList )
```

[8] `Layer` is located in the JCop standard library package `jcop.lang`.

```java
public void main() {
  B b = new B();
  Alpha alpha = new Alpha();
  Alpha alpha2 = new Alpha();

  with (alpha) {
    b.y();
  }

  with (alpha, expr()) {
    b.y();
  }

  with (alpha) {
    with (new Beta()) {
      b.y();
    }
  }

  with (alpha, alpha, alpha2) {

    without (alpha) {
      b.y();
    }
    withoutall(Alpha.class) {
      b.y();
    }
  }
}

private Layer expr() {
  return new Beta();
}
```

$(\mathsf{base})\ \ \mathit{with}\ \ (alpha) \mapsto (alpha, \mathsf{base})$

$(\mathsf{base})\ \ \mathit{with}\ \ (alpha, beta) \mapsto (beta, alpha, \mathsf{base})$

$(\mathsf{base})\ \ \mathit{with}\ \ (alpha) \mapsto (alpha, \mathsf{base})$
$(alpha, \mathsf{base})\ \ \mathit{with}\ \ (beta) \mapsto (beta, alpha, \mathsf{base})$

$(\mathsf{base})\ \ \mathit{with}\ \ (alpha, alpha, alpha2)$
$\mapsto\ \ (alpha2, alpha, alpha, \mathsf{base})$

$(alpha2, alpha, alpha, \mathsf{base})\ \ \mathit{without}\ \ (alpha)$
$\mapsto\ \ (alpha2, \mathsf{base})$

$(alpha2, alpha, alpha, \mathsf{base})\ \ \mathit{without}\ \ (alpha, alpha2)$
$\mapsto\ \ (\mathsf{base})$

Listing 10: Explicit layer compositions and their corresponding composition functions.

> *LayerCompositionFunction : one of*
> `with without withoutall`

The `with` block statement can be used in method bodies. It contains an argument list of layer type expressions denoting the layers to be activated. More precisely, expressions of type `Layer`[9], `Iterable<Layer>`, or `Layer[]` are valid arguments; the usage of any other type will cause a compile time exception. If all `with` arguments are evaluated to an empty list (or `null`), no layer is activated.

The specified layers are only active for the *dynamic extent* of the `with` block. That is, for the control flows of any expression within the block. This implies that the activation of a particular layer is confined to the threads in which the layer was explicitly activated. Layer activation does not propagate to new threads; they start with no layers being active[10].

[9] `jcop.lang.Layer`

[10] Section 4.3.1 describes how layer composition could be easily transfered to other threads using

```java
public void main() {
  Composition comp = Composition.current();
  Composition newComp = comp.withLayer(new Alpha());
  with(newComp.getLayers()) { ... }
}
```

Listing 11: Reflective layer activation

```java
public void m() {
  with(new Alpha()) {
    ...
    final Composition compOfOtherThread = Composition.current();
    new Thread() {
      public void run() {
          with(compOfOtherThread.getLayers()) {
            ...
          }
      }
    }.start();
}
```

Listing 12: Using reflection to instrument new threads with a layer composition.

4.3 Reflective Layer Composition

The constructs presented so far support most common scenarios for layer composition. For situations requiring special reasoning about layers and their composition, JCop provides a reflection API. It gives access to inspect and manipulate layers, their composition and their partial methods at run-time. In addition, JCop provides *layer-based composition*—the ability of layers to decide about their activation .

4.3.1 Reflection API

The complete reflection API is documented in Appendix 6. Its class `Composition` represents a layer composition and contains the ordered list of layers. It provides access and navigation through the composition's layers. The methods `withLayers`, `withoutLayers`, and `withoutAllLayers` correspond to the composition functions presented in the previous sections and return a new instance representing the modified composition. Note that the methods do not activate the composition object. For activation, the composition's layers can be passed to a composition function. In Listing 11, the run-time composition is accessed via the `Composition` interface. Using the `withLayer` method, a new composition object containing the old composition plus an instance of `Alpha` is generated. An array containing the composition's layers

meta programming

```
public class Display {
  public void render() { ... }
}

public layer DayDisplay {

  public Composition onWith(Composition current) {
    Composition comp = current.withoutAllLayer(NightDisplay.class);
    if (!comp.contains(Display3D.class)) {
      comp = comp.withLayer(new Display3D());
    }
    return comp;
  }
  public Composition onWithout(Composition current) {
    return current.withoutAllLayer(Display3D);
  }
  public void render() { ... }
}

public layer NightDisplay {
  public Composition onWith(Composition current) {
    return current.withoutAllLayer(DayDisplay.class);
  }
  public void Display.render() { ... }
}

public layer Display3D { ... }
```

Listing 13: Implicit layer-based activation.

is then used as an argument of the `with` statement.

A new thread that is started in this dynamic extend will not inherit the composition of its creating thread. However, using reflection we can easily initialize the new thread with the old layer composition, as demonstrated in Listing 12. First, we assign the layer composition list of the parent thread to a final variable of type `Composition`. Then, we override the method `run` in the new thread. In this method, we access the layers of the composition variable using `getLayers`. Finally, we activate these layers in the new thread.

4.3.2 Layer-based Composition

Structural relationships between layers is addressed by JCop's layer inheritance mechanisms. In addition, some application-specific dependencies may impact layer composition. For example, the use of a layer may prevent or cause the activation of another layer. JCop achieves such *layer-based composition* by an event-handler mechanism that allows layers to manipulate the run-time composition on activation and deactivation. For that purpose, the interface of `jcop.lang.Layer` —the implicit superclass of all layers—provides the two event handler methods `onWith` and `onWithout` that can be overwritten by concrete layers. The handlers are called for explicit and declarative layer composition. However, they are not called for reflective layer activation, which could lead to infinite loops. The handlers are right after layer activation and right before layer deactivation. The current composition is passed as an argument to the method so that it can be analyzed and manipulated using reflection as described in the previous paragraph. The handler methods return a composition object that is activated instead of the input composition.

We give an example of the use of layer-based activation by expressing layer dependencies such as (mutual) exclusions and inclusions. The example in Listing 13, implements two layers, `DayDisplay` and `NightDisplay` for a GPS device. To assure that instances of both layers are never active at the same time (mutual exclusion), we override `onWith`. On activation of one layer, its composition handler method takes care that any instance of the other one is removed from the list. We can also include other layers on activation of a specific layer. For example, the layer `DayDisplay` implicitly activates `Display3D` on activation.

For convenience, `Composition` provides the methods `requires`, `excludes`, `excludesAll` `weakRequires`, `weakExcludes`, and `weakExcludesAll` with which a layer can express its dependencies. The method `requires` checks if the required layer is part of the current composition. If not, the application throws a `CompositionException` that may be used to solve the conflict. The method `weakRequires` also checks for the presence of the required layer, but instead of throwing a `CompositionException`, it implicitly activates the layer. The other methods, `excludes` and `weakExcludes` work in a similar manner.

4.3.3 Object-based Activation

Objects can implicitly activate layers on execution of their layered methods. That is, even if the composition does not contain a layer for a layered method $A.x$, on execution of $A.m$, A can decide to activate a layer to execute a partial method of x. As with the layer event handlers of the previous subsection, objects can implement the event handler `onLayeredExecution` provided by the interface `LayerProvider`: Listing 14 shows an extreme example in which an object prohibits any further modification by layers[11]:

[11] In most cases, declaring the methods to be protected as `final` would also serve this purpose and might be the more elegant solution. Final methods cannot be adapted by partial methods

```
interface LayerProvider {
  public Composition onLayeredExecution(Composition current);
}

class UnadaptableClass implements LayerProvider {
  public Composition onLayeredExecution(Composition current) {
    return current.withoutLayer(current.getLayers());
  }
}
```

Listing 14: Use of the `LayerProvider` interface.

5 Declarative Compositions

JCop's declarative layer composition is implemented by a domain-specific aspect language. Its join point model consists solely of method executions. These join points can be specified by pointcuts. Syntactically, declarative compositions consist of two parts, a pointcut part and an advice composition part. The pointcut part is a logic expression consisting of built-in and named pointcuts, see Subsection 5.1. Subsection 5.2 discusses the evaluation of `when` pointcuts. The advice composition contains a sequence of `with` and/or a `without` operators, see Subsection 5.3.

5.1 Layer Composition Pointcuts

Layer composition pointcuts can be composed using the logic operators *and* ('`&&`'), *or* ('`||`') and *not* ('`!`'), like pointcuts in AspectJ [12] and predicates in Prolog [13]. JCop provides four *built-in pointcuts*, `on`, `when`, `this`, and `args`, and the declaration of *named pointcuts*.

Built-in Pointcuts

The `on` pointcut can describe method executions at which layers should be composed. The respective method is specified by its signature. The example in Listing 15 presents an `on` pointcut that surrounds the execution of `A.x` and `A.y` with the activation of `Alpha` (Lines 1–3). The same behavior is shown on the right using explicit activation (Lines 10, 11).

The `when` pointcut allows for a more implicit description of composition time independent of the actual execution in the main control flow. It is useful for applications in which context activation depends on the change of a specific property (such as the state of a sensor) that can be evaluated by a boolean expression. Whenever this expression evaluates to true, the layer composition is applied. Listing 16 presents a `when` pointcut that activates `Alpha` depending on location data.

```
on (int pckg.A.x()) ||
on (void pckg.A.y(int)) :
  with(new Alpha());
```

```
package pckg;

public class A {
  public int x() { with(new Alpha()) { ... } }
  public void y(int i) { with(new Alpha()) { ... } }
}
```

Listing 15: Modular declarative *(left)* vs. redundant explicit composition statements *(right)*.

```
when (Sensor.inLibrary()) :
  with(new SilenceMode());

layer SilenceMode {
  public void PhoneAlert.incomingCall() {
    // vibrate
  }
  public void Display.update() {
    // discrete message
  }
}

class Sensor {
  public static boolean inLibrary() {
    /* checks GPS and RFID data */
  }
}
```

```
class PhoneAlert {
  public void incomingCall() {
    // ring
    display.update();
  }
}

class Display {
  public void update() {
    // bright blinking
  }
}
```

Listing 16: A `when` predicate that handles phone recomposition.

The two remaining pointcuts `this` and `args` help to further specify the join points collected by `on` and `when` but do not bind new join points. For example, the following pointcut uses an reference of type `A` to restrict the join points collected by `on` to instances of `a`.

```
A myA = A();
...
on (int pckg.A.x()) && this(a)
```

If the type of `a` does not match the type of the method, the pointcut expression returns an empty join point set.

Named Pointcuts Pointcut expressions may become complex and hard to comprehend. For a better modularization, they can be explicitly declared by a named pointcut declaration. Named pointcut declarations are treated as a member of their enclosing context class, see Section 5.4. As a member, it may have an access modifier such as `public`, `protected`, `private`, `abstract`, or `final`.

Syntactically, named pointcuts in JCop are similar to their counterparts in AspectJ. However, named pointcuts in JCop do not have parameters because they do not

```
public @interface Evaluate {
   EvalPolicy value() default  EvalPolicy.OncePerCflow ;
   public enum EvalPolicy {OncePerCflow, UntilTrue, UntilFalse, Always}
}

@Evaluate(EvalPolicy.UntilTrue)
when (Sensor.inLibrary()) : with(new SilenceMode());
```

Listing 17: The `Evaluate` annotation can specify the evaluation policy of `when` pointcuts.

pass variables to their advice block, see Subsection 5.3. For example, using a named pointcut, the layer composition declaration of Listing 15 can separate the pointcut specification from the layer composition:

```
pointcut relevantMethods : on (int pckg.A.x()) && on (void pckg.A.y(int));
relevantMethods : with(new Alpha());
```

5.2 Evaluation of `when` Pointcuts

The `when` pointcut expressions are evaluated every time a method invocation is potentially dispatched to a layer involved in the composition, i.e, at every execution of a layered method.

In general, it is hard to predict the actual number layered method calls and with that the number of pointcut evaluations in a program execution. Therefore, the pointcut expressions are considered to be side-effect free[12]. In this example, the pointcut evaluates the boolean method `inLibrary` on every invocation of `incomingCall` and `update` because these methods are affected by the layer composition specified in the advice: on activation of `SilenceMode` their partial methods become active.

In order to guarantee that a dynamic extent is executed with a consistent layer composition, we impose an additional restriction to the `when` evaluation. Figure 5 (left) illustrates an issue with multiple evaluations of `when` within one control flow for the library example in Listing 16. On invocation of `incommingCall`, the pointcut checks if the layer should be activated. The sensor returns false, therefore the base method of `incommingCall` is executed. During the execution of `incommingCall`, the method `update` is called. This causes again the evaluation of the pointcut because the layer `SilenceMode`—which should be activated by the advice—provides also a partial method for `update`. This causes inconsistent behavior because the phone alert is executed in the default mode, but the display is rendered for a discreet

[12] A similar restriction is specified for expressions used in guard predicates in the ObjectTeams language [10].

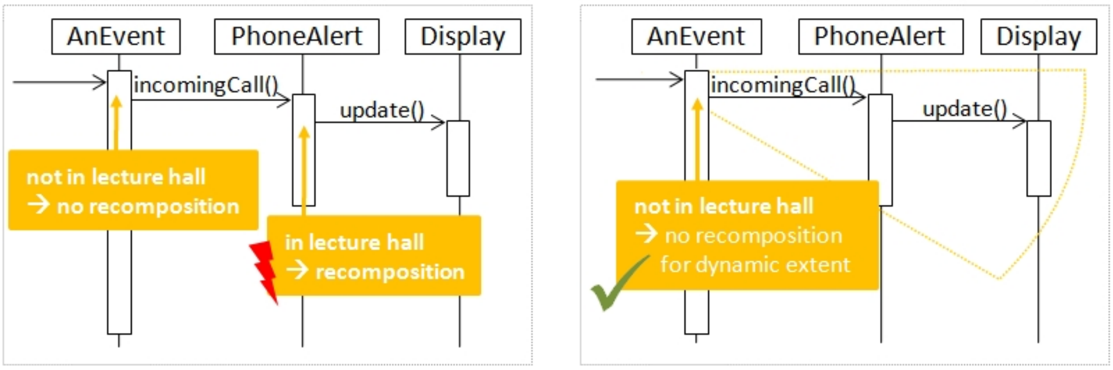


Figure 5: Composition consistency for pointcut-based layer adaptation.

notification. JCop prevents such inconsistent behavior by ensuring that, once `when` is evaluated, it is not re-evaluated within the dynamic extent originating from this evaluation, as depicted in Figure 5 (right). This strategy conforms to the original context-oriented programming model: once a composition has been activated, it is consistent and valid until its `with` block terminates.

However, if another evaluation strategy is explicitly desired, the `when` pointcut can be annotated with `@Evaluate`. The annotation expects a value of the enumeration `EvalPolicy` that contains four elements. The element `OncePerCflow` represents the default behavior explained in the previous paragraph (i.e., no annotation is specified). If instead the evaluation within a control flow should be continued until the condition reaches either `true` or `false`, the annotation can specify `UntilTrue` or `UntilFalse`. For a continuous evaluation at any method execution, the evaluation policy is set to `Always`. This may be useful if the expression to be evaluated has side effects that should be always executed. Listing 17 shows the declaration of `jcop.lang.Evaluate` and a specification of a `when` pointcut to be executed at any method. Note that a composed pointcut expression can only specify one evaluation strategy (i.e., one `@Evaluate` annotation).

5.3 Composition Advice

A composition advice consists of a comma separated list of `with`, `without`, or `withoutall` operations that are applied to the join point consecutively. The composition block is only executed if the pointcut condition matches at least one join point. There are two different reasons for a pointcut to match. Either, a method is executed that is declared by an `on` pointcut. Or, the expression of a `when` pointcut evaluates to true. The layers specified by the composition operations are be applied to the composition in the order of their declaration.

```
1 public contextclass LibraryContext {
2   private SilenceMode silence = new SilenceMode();
3
4   when (inLibrary()):
5     with(new SilenceMode());
6
7   public boolean inLibrary() {
8     /* checks GPS and RFID data */
9   }
10 }
11
12 public class Main {
13   void main() {
14     LibraryContext library = new LibraryContext();
15     library.activate();
16   }
17 }
```

Listing 18: A context class declaration.

5.4 Context Class Declaration

Declarative compositions and named pointcuts, presented in the previous sections, are enclosed by a *context class declaration*, a special class declaration that must contain at least one composition declaration. In addition, context class declarations can contain any class member declarations and named pointcut declarations. Context class declarations use the keyword `contextclass` instead of `class`.

Listing 18 shows a context class for our previous library example. Besides the composition declaration, it contains an auxiliary method to evaluate context change and a private layer field. Declarative compositions are evaluated at instance level. Hence, multiple instances of a context class can influence the layer composition. The evaluation of the predicates can be controlled by two means. First, like layers, context classes can be declared to be *static active* by a modifier. In this case, a *singleton* instance of the context class (using the default constructor) will be automatically created on static initialization. Second, context class instances can be *deployed* and *undeployed*, in a similar way to dynamic aspects languages, such as CaesarJ [4]). The interface of the implicit super type `jcop.lang.ContextClass` provides the deployment API shown in Appendix 6.2.1. Context class activation is thread-local but not bound to a dynamic extend like layer activation. Also note that context activation does not immediately cause any changes to the composition but registers composition statements for future computations. Line 15 of Listing 18 shows a dynamic activation of the context class `LibraryContext`.

6 Appendix

6.1 ENBF of the JCop Language Extension to Java

We present the syntax production rules in the following sections using the grammar notion of the Java Language Specification, third edition [9]. Terminals are shown in `fixed font`, non-terminals are shown in *italic*. Each line of a grammar definition represents one alternative right-hand side. The subscripted suffix '$_{opt}$' after elements denotes an optional symbol. The words '*one of*' after a grammar definition header starts an abbreviation that declares each of the following terminal symbols as alternative definitions. Some JCop production rules extend existing Java production rules, which we mark with underlines and abbreviate their definition using '...'.

6.1.1 Layers and Partial Methods

6.1.2 Layer Declaration

ClassDeclaration :
  ...
  LayerDeclaration
  ContextClassDeclaration

LayerDeclaration :
  LayerModifiers$_{opt}$ `layer` *Identifier* *Super*$_{opt}$ *Interfaces*$_{opt}$
    LayerBody

LayerBody :
  `{` *LayerBodyDeclarations*$_{opt}$ `}`

LayerBodyDeclarations :
  LayerBodyDeclaration
  LayerBodyDeclarations *LayerBodyDeclaration*

LayerBodyDeclaration :
  ClassMembersDeclaration
  LayerMemberDeclaration

Layer Modifiers

LayerModifiers :

$LayerModifier$
$LayerModifiers\ LayerModifier$

$LayerModifier\ :\ one\ \ of$
$\underline{ClassModifier}$ staticactive

Open Layer Declaration

$\underline{ClassBodyDeclaration}\ :$
$...$
$OpenLayerDeclaration$

$OpenLayerDeclaration\ :$
layer $\underline{Identifier}$ { $OpenLayerBodyDeclaration_{opt}$ }

$OpenLayerBodyDeclaration\ :$
$\underline{ClassMemberDeclaration}$
$LocalPartialMethodDeclaration$

Partial Method Declaration

$LayerMemberDeclaration\ :$
$PartialMethodDeclaration$

$PartialMethodDeclaration\ :$
$PartialMethodHeader\ \underline{MethodBody}$

$PartialMethodHeader\ :$
$PartialMemberModifiers_{opt}\ \underline{ResultType}\ LayerMethodDeclarator\ \underline{Throws}_{opt}$

$LayerMethodDeclarator\ :$
$FullQualifiedName$ ($\underline{FormalParameterList}_{opt}$)

$FullQualifiedName\ :$
$\underline{TypeName}\ .\ \underline{Identifier}$

$PartialMemberModifiers\ :$
$PartialMemberModifier_{opt}\ Modifiers$

$PartialMemberModifier\ :\ one\ \ of$
before after final abstract

$\underline{Expression}\ :$
$...$

```
  ProceedExpression
  SuperProceedExpression

Access :
  ...
  thislayer
  superlayer

ProceedExpression :
  proceed ( ArgumentList )

SuperProceedExpression :
  superproceed ( ArgumentList )

LocalPartialMethodDeclaration :
 LocalPartialMethodHeader MethodBody

LocalPartialMethodHeader :
  PartialMemberModifiers_opt ResultType MethodDeclarator Throws_opt
```

6.1.3 Layer Composition

Explicit Composition Statement

```
StatementWithoutTrailingSubstatement :
  ...
  LayerCompositionStmt

LayerCompositionStmt    :
  LayerCompositionHeader Block

LayerCompositionHeader :
  LayerCompositionFunction( ArgumentList )

LayerCompositionFunction : one of
  with without withoutall
```

Declarative Composition

```
DeclarativeComposition :
  CompositionPointcuts : CompositionSpecification ;
```

```
CompositionSpecification :
  LayerCompositionHeader
  LayerCompositionHeader , CompositionSpecification
```

Pointcut Declaration

```
CompositionPointcuts : one of
  CompositionPointcut EmbracedComposition LogicComposition

CompositionPointcut : one of
  BasicPointcuts NamedPointcutAccess

BasicPointcuts : one of
  onPointcut whenPointcut thisPointcut argsPointcut


PointcutPattern :
  Modifiers_opt ResultType LayerMethodDeclarator

LogicPointcut : one of
  NegatedPointcut AndPointcut OrPointcut

EmbracedPointcut:
  ( CompositionPointcuts )

AndPointcut :
  CompositionPointcuts && CompositionPointcuts

OrPointcut :
  CompositionPointcuts || CompositionPointcuts

NegatedPointcut :
  ! CompositionPointcuts

NamedPointcutDeclaration :
  pointcut Identifier : CompositionPointcuts ;

Access :
  ...
  NamedPointcutAccess

NamedPointcutAccess :
  Identifier

onPointcut :
    on ( PointcutPattern )

  whenPointcut :
    when ( Expression )

  thisPointcut :
    this ( Expression )

  argsPointcut :
    args ( ArgumentList )
```

6.1.4 Context Class Declaration

```
  ClassDeclaration :
    ...
    LayerDeclaration
    ContextClassDeclaration

  ContextClassDeclaration :
    ContextClassModifiers_opt contextclass Identifier
      ContextClassBodyDeclaration

  ContextClassBodyDeclaration :
    { ContextClassMembers }

  ContextClassMembers :
    ContextClassMember
    ContextClassMembers ContextClassMember

  ContextClassMember : one of
    DeclarativeComposition NamedPointcutDeclaration ClassMember

  ContextClassModifiers :
    ContextClassModifier
    ContextClassModifiers ContextClassModifier

  ContextClassModifier : one of
    ClassModifier staticactive
```

6.1.5 Additional Syntax Elements

Unless Statement

The `unless` statement is taken from the Ruby language and is syntactic sugar for negative `if` conditions.

```
UnlessThenStatement :
  unless ( Expression ) Statement

UnlessThenElseStatement :
  unless ( Expression ) StatementNoShortIf else Statement
```

The Expression must have type `boolean` or `Boolean`, or a compile-time error occurs. Executes code if conditional expression is false. If the conditional is true, code specified in the else clause is executed.

If and Unless Modifier

JCop shares another feature with Perl and Ruby. Statement modifiers let you tack conditional statements onto the end of a normal statement.

```
Statement ::=
  ... |
  ConditionalModifier

ConditionalModifier ::=
  IfModifier |
  UnlessModifer

IfModifier ::=
  StatementExpression if Expression ;

UnlessMofifier::=
  StatementExpression unless Expression ;
```

6.2 Libraries and APIs

In the following, we document the reflective API that is part of the JCop core and the context query library that we developed during the development of our

WhenToDo application.

6.2.1 Reflection API

The reflective API is integrated with Java's API `java.lang.reflect`. It consists of three classes of the `jcop.lang` package, namely `Layer`, `Composition`, and `PartialMethod`. `Layer` provides reflective access to its partial method definitions. `Composition` objects allow access to their layers and the (de-)activation of layers. `PartialMethod` is the meta-class of partial methods, corresponding to Java's `java.lang.reflect.Method` class. As `Method`, it inherits from `AccessibleObject` and implements the `Member` interface, which are both defined in the package `java.lang.reflect`, see Figure 6.

jcop.lang.Layer

```
public Composition onWith(Composition current)
```

Returns the current thread-local composition. The method can be overridden for specific composition handling.

```
public Composition onWithout(Composition current)
```

Returns the current thread-local composition. The method can be overridden for specific composition handling.

```
public static Composition onWithoutAll(Composition current)
```

Returns the current thread-local composition. The method can be overridden for specific composition handling.

```
public void includes(Layer toBeIncluded, boolean stopOnConflict)
```

Specifies that on layer activation `toBeIncluded` must be part of the composition. The rule is checked right after the activation of the layer by the default implementation of `onWith` in `jcop.lang.Layer`. If `stopOnConflict` is `true`, `onWith` will throw a `CompositionException`. If `stopOnConflict` is `false`, `onWith` will activate `toBeIncluded`.

```
public void excludes(Layer toBeExcluded, boolean stopOnConflict)
```

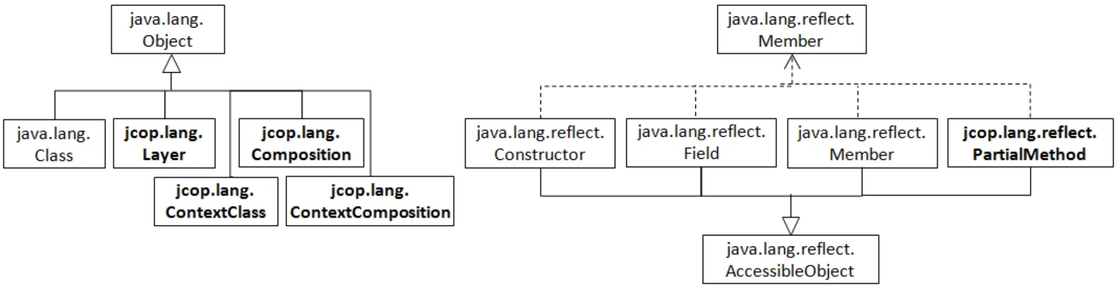


Figure 6: Extension of the Java reflection API.

Specifies that on layer activation `toBeExcluded` must not be part of the composition. The rule is checked right after the activation of the layer by the default implementation of `onWith` in `jcop.lang.Layer`. If `stopOnConflict` is `true`, `onWith` will throw a `CompositionException`. If `stopOnConflict` is `false`, `onWith` will remove `toBeExcluded` from the composition.

`public` `Composition getComposition()`

Returns the enclosing layer composition

`public boolean` `isActive()`

Returns `true` if the layer is activated in the current thread.

`public boolean` `providesPartialMethodFor(String)`

Determines if the layer provides a partial definition for a method with signature represented by the parameter

`public` `PartialMethod[] getPartialMethods()`

Returns an array of `PartialMethod` objects reflecting all the partial methods provided by the layer.

`public` `PartialMethod getPartialMethod(String)`

Returns a `PartialMethod` object representing a partial method of the layer with the signature specified by the parameter.

jcop.lang.ContextClass

`public void` `deploy()`

Activates the context class. Multiple activation is ignored.

`public void` `undeploy()`

Deactivates the context class. Multiple deactivation is ignored.

`public boolean` `isDeployed()`

Returns true it the context class is active.

jcop.lang.Composition

`public static` `Composition current()`

Returns the current thread-local composition.

`public` `Composition withLayers(Layer... layers)`

Activates a layer in the current composition. Returns a clone of the old composition before the activation.

`public` `Composition withoutLayers(Layer... layers)`

Deactivates a layer in the current composition. Returns a clone of the old composition before the activation.

`public` `Composition withoutAllLayers(Class<Layer>... layers)`

Deactivates a all instances of a layer type in the current composition. Returns a clone of the old composition before the activation.

`public` `Layer firstLayer()`

Returns the first layer of the composition.

`public Layer next(Layer currentLayer)`

Returns the successor of currentLayer in the composition. Returns null, if currentLayer is not part of the composition.

`public boolean contains(Layer aLayer)`

Returns true if the layer is part of the composition.

`public boolean contains(Class<Layer> aLayer)`

Returns true if the composition contains at least one instance of the layer.

`public Layer[] getLayers()`

Returns the composition's layers as array.

jcop.lang.ContextComposition

`public static ContextComposition current()`

Returns the thread-local context composition.

`public ContextClass[] getContextClasses()`

Returns the deployed context class instances.

jcop.lang.PartialMethod

`public Layer getDefiningLayer()`

Returns the layer defining this partial method

`public Class getDeclaringClass()`

Returns the declaring class of the partial method

`public Class[] getExceptionTypes()`

Returns an array of the exception types

`public String getName()`

Returns a string representation of that method

`public Class getReturnType()`

Returns the return type of the method

`public int getModifiers()`

Returns the Java language modifiers for the method represented by this Method object, as an integer

`public Object invoke(Object target, Object... args)`

Invokes the underlying partial method on the specified object with the specified parameters

jcop.lang.ILayerProvider (Interface)

`public Composition onLayeredExecution(Composition current)`

> Objects can implicitly activate layers on execution of their layered methods by implementing this event handler method. Therefore, even if the composition does not contain a layer for a layered method $A.x$, on execution of $A.m$, A can decide to activate a layer to execute a partial method of x.

6.2.2 Context Query Library

Our query library alleviates the access to the CMS. It supports executing context queries and defining actions - for example, layer activation - to be taken on context change. In the following, we describe the most important API objects and methods.

Query Language Overview The query language is syntactically inspired by OCL (*Object Constraint Language*) and supports logic reasoning about context sources. A query depends on *context sources* that provide context information and meta-data. The context information is retrieved from sensors, local applications or external services. Context sources are represented by RDF (Resource Description Framework) which represents contexts and their meta-data as object-graphs. Contexts are queried by predicates over the Java RDF types. We can refer to the set of instances of a class with its class name `C`. The language provides a number of operations on context sets, aligned to the OCL names: `select`, `reject`, `forAll`, `exists` and additionally the `one` operation. The following expression selects all context instances bound by the expression `e` which fulfill the condition c:

```
e->select(c)
```

Properties on a list of contexts are mapped to each list member individually, meaning for all list elements for which the property exists, the property is added to the returned list. So the following expression matches all properties `p` of instances of `C` fulfilling `c`:

```
C->select(c).p
```

The `one` operation is special to the CQL compared to OCL. It binds one context instance at a time and the CQL backtracks over all elements of a list:

```
e->one(c)
```

The condition `c` is evaluated in the context of the queried type. If the property has a range (`rdfs:range`) of rdf container or collection `v` will be bound to the array of all elements, otherwise it will backtrack over all defined properties, e.g. in case `Contact->one().name` over all defined `name` properties. The instance of the queried class may be named; in the following example, the current contact instance is bound to `c`.

```
Contact->one( c | c.firstName=value1 && c.lastName=value2 ) {...}
```

All logic variables must be declared at the beginning of the expression enclosed in parenthesis and separated by a colon from the query. Variables and expression can be unified via the unification operator '='. For example, the following query binds the variable `c` to a `Contact` object and the CQL backtracks over all contact objects in the context factbase.

```
(Contact c): c = Contact->one()
```

Context query sources are accessed by the following expression.

```
var = SensorTypeName->methodName({param1 = value1,...})
```

Continuing the nearby contacts example, a query returning information about contacts within a maximum distance of 100 meters can be written as:

```
(Nearby[] nc):
   nc = NearbyService->nearby({maxDistance=100})
```

Universally quantified statements over context lists are written as `e->forAll(c)` and the existential quantification is written as `e.exists(c)`. The language supports all primary logic operations (*and*, *or*, *not*). The following conditions ascertains that all contacts from the given list cs are nearby and that their distance to the current position is less than the value max.

```
(Contact[] cs, int max) : cs->forAll(c | NearbyService->
   nearby({maxDistance=max})->exists(c.email=email))
```

Library API

jcop.cms.ContextQuery `public ContextQuery(ContextRequest, String, String)`

A query object is instantiated with its context type schema, a default namespace, and a string representation of the CQL expression.

`public boolean evaluate()`

A query can be executed synchronously and will immediately return a Boolean value whether the context is constituted or not.

`public void evaluate(IContextHandler)`

Queries can also be executed asynchronously. In that case, an `IContextHandler` object is passed to the query's evaluation method that is called by the CMS on context entry and exit.

`public ResultSet getResultSet()`

The variable bindings of the last executed query are represented by a `ResultSet` that holds a list (for each solution found via backtracking) containing maps of key-value pairs. In addition, `ResultSet` provides some auxiliary methods such as `boolean isEmpty()`.

`public void addLayers(Layer[])`

Layers can be associated with a query, for example, to make them accessible to the `IContextHandler` callback methods.

jcop.cms.IContextHandler The `IContextHandler` interface is frequently used if queries are evaluated asynchronously.

`public void IContextHandler.onContextEntry/onContextExit(Layer[])`

The callback methods are activated on context change and can be used for implementing any kind of reaction to the new state. They are parameterized with `Layer` objects if they have been associated with the query.

6.3 Compiler Options

In the following, we describe the usage of the JCop compiler that is able to compile programs of the JCop and Java programming language. The compiler implements the language as defined in this report. The compiler is an extension to the JastAddJ [7] Java compiler that is implemented based on the JastAdd compiler framework.

6.3.1 Synopsis

```
jcopc [Options] [file]
```

6.3.2 Description

The `jcopc` command compiles JCop and Java source, producing `.class` files compliant with any Java VM (1.5 or later). The argument after the options specifies the source file to compile. Source files are specified by their full qualified name (package name + type name), separated by a dot ("."). The command `jcopc` accepts source files with either the `.java` extension or the `.jcop` extension.

6.3.3 Options

`-classpath <path>`
Specifies where to find user class files and annotation processors.

`-sourcepath <path>`
Specifies where to find input source files. Only required if the sources are not located in the working directory. Example:

```
jcopc -sourcepath src myPckg.MyMainClass}
```

`-d <directory>`
Specifies where to place generated class files. Example:

```
jcopc -d bin -sourcepath src myPckg.MyMainClass
```

`-sourcedump <path>`
Dumps Java source files of the compiled classes into the specified folder. Example:

```
jcopc -sourcedump dump myPckg.MyMainClass
```

`-agg <path>`
Generates a file containing an AGG graph representation of the program's AST. Example:

```
jcopc -agg output/agg myPckg.MyMainClass
```

`-ctl`
Logs JCop-specific messages about what the compiler is doing. Example:

```
jcopc -ctl myPckg.MyMainClass
...
> copying PartialMethod.java
> to src\jcop\lang
> ..done
...
> compiling:...src\de\uni_potsdam\hpi\swa\Widget.jcop ..done
> compiling:jcop\lang\Composition.java ..done
...
> ..done
> compile and weave auxilliary aspect  bin\jcop\lang\JCopAspect.aj
> ..done
> compiled in 3723 millis
```

-rtl
Logs layer activation and composition information at runtime. Example:

```
jcopc  -rtl  myPckg.MyMainClass
...
>  INFO:  accessing  base  method  of  getBMI
>  INFO:  accessing  method  getBMI  of  layer  Visualization
...
```

-help
Prints a synopsis of standard options.

-aspectinfo
Prints messages about aspect weaving.

-version
Prints version information.

-xml-outline-path <path>
Generates an outline in XML format. For each compilation unit, a corresponding XML file is generated.

-class-in-layer-outline
Generates a layer cross-reference view in XML format that outlines which methods are adapted by a layer.

6.3.4 Underlying Java Command

If you encounter any problems using jcopc, you may use the plain Java command. For instance, the command

```
jcopc -classpath bin -ctl MyApplication
```

is equivalent to:

```
java -jar -ea "%JCOP_HOME%\jcop.jar" -classpath bin -ctl MyApplication
```

6.4 Launcher Options

The JCop launcher instruments the Java launcher with some libraries that are required by

6.4.1 Synopsis

```
jcop [Options] class [arguments]
```

6.4.2 Description

The jcop command launches a JCop or Java application. It does this by starting a Java runtime environment (instrumented with the AspectJ weaver), loading a specified class, and invoking that class's main method. The method declaration must look like a plain Java main method:

```
public static void main(String args[]) {...}
```

The method must be declared public and static, it must not return any value, and it must accept a String array as a parameter. By default, the first non-option argument is the name of the class to be invoked. A fully-qualified class name should be used.

The argument after the options specifies the class file to compile and by its full qualified name (package name + type name), separated by a dot ("."). The `jcop` command accepts Java bytecode files with the class extension.

6.4.3 Options

All standard Java options are accepted by the JCop launcher.

6.4.4 Underlying Java Command

If you encounter any problems using jcop, you may use the plain java command. For example, in Windows, the command

```
jcop -classpath "bin;lib\mylib\" MyApplication my args
```

is equivalent to:

```
java -classpath "%JCOP_HOME%\aspectjweaver.jar;bin;lib\mylib\"
     "-Djava.system.class.loader=org.aspectj.weaver.loadtime.WeavingURLClassLoader"
     "-Daj.class.path=%ASPECTPATH%;%CLASSPATH%"
     "-Daj.aspect.path=%ASPECTPATH%"
     MyApplication my args
```

6.5 Limitations of the JCop Compiler

Use of the Default Package Layer types cannot be declared in the default package. However, classes of the default package can import and use layers.

Nested Class Support Layers cannot be declared and activated within nested and anonymous classes:

```
public  class  MyClass  {
    class  MyInnerClass  {
      void  m()  {  ...  }
      layer  MyLayer{  ...  }  //  this  wonŠt  work
    }
    void  myMethod()  {
      new  MyNestedClass()  {
        layer  MyLayer  {  ...  }  //  this  wonŠt  work
    }
  }
```

Reserved Keywords In addition to Java's keywords the following keywords are introduced by JCop and cannot be used as identifiers. This might require identifier renaming in existing Java programs that should be compiled by JCop.

```
after, before, contextclass, layer, on, when,
this, args, proceed, with, without, withoutall
```

Bibliography

[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, and Hidehiko Masuhara. Improving the Development of Context-dependent Java Applications with ContextJ. In *International Workshop on Context-Oriented Programming*, COP'09, pages 1–5, New York, NY, USA, 2009. ACM Press.

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-based 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.

[4] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of CaesarJ. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, 3880:135–173, 2006.

[5] Curtis Clifton, Todd D. Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.

[6] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS'05, pages 1–10, New York, NY, USA, 2005. ACM Press.

[7] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA'07, pages 1–18, New York, NY, USA, 2007. ACM Press.

[8] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The 3rd Edition.* Addison-Wesley Professional, Boston, MA, USA, 2005.

[10] S. Herrmann, C. Hundt, and M. Mosconi. ObjectTeams/Java Language Definition - version 1.0. Technical Report 3, 2007.

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

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented Programming. In *Proceedings 11th European Conference on Object-Oriented Programming, ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Berlin, Heidelberg, Germany, 1997. Springer-Verlag.

[13] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, Germany, second edition, 1987.

[14] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communication ACM*, 3:184–195, April 1960.

[15] Tobias Rho, Malte Appeltauer, Stephan Lerche, Armin B. Cremers, and Robert Hirschfeld. A Context Management Infrastructure with Language Integration Support. In *Proceedings of the Third International Workshop on Context-Oriented Programming.*, COP'11, pages 1–6, New York, NY, USA, 2011. ACM Press.

[16] Christopher Schuster, Malte Appeltauer, and Robert Hirschfeld. Context-oriented Programming for Mobile Devices: JCop on Android. In *Proceedings of the Third International Workshop on Context-Oriented Programming, COP'11*, pages 1–6, New York, NY, USA, 2011. ACM Press.

[17] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby. The Pragmatic Programmer's Guide.* Pragmatic Programmers, 2004.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
58	978-3-86956-192-9	**MDE Settings in SAP: A Descriptive Field Study**	Regina Hebig, Holger Giese
57	978-3-86956-191-2	**Industrial Case Study on the Integration of SysML and AUTOSAR with Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Stefan Neumann, Sebastian Wätzoldt
56	978-3-86956-171-4	**Quantitative Modeling and Analysis of Service-Oriented Real-Time Systems using Interval Probabilistic Timed Automata**	Christian Krause, Holger Giese
55	978-3-86956-169-1	**Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium**	Peter Tröger, Andreas Polze (Eds.)
54	978-3-86956-158-5	**An Abstraction for Version Control Systems**	Matthias Kleine, Robert Hirschfeld, Gilad Bracha
53	978-3-86956-160-8	**Web-based Development in the Lively Kernel**	Jens Lincke, Robert Hirschfeld (Eds.)
52	978-3-86956-156-1	**Einführung von IPv6 in Unternehmensnetzen: Ein Leitfaden**	Wilhelm Boeddinghaus, Christoph Meinel, Harald Sack
51	978-3-86956-148-6	**Advancing the Discovery of Unique Column Combinations**	Ziawasch Abedjan, Felix Naumann
50	978-3-86956-144-8	**Data in Business Processes**	Andreas Meyer, Sergey Smirnov, Mathias Weske
49	978-3-86956-143-1	**Adaptive Windows for Duplicate Detection**	Uwe Draisbach, Felix Naumann, Sascha Szott, Oliver Wonneberg
48	978-3-86956-134-9	**CSOM/PL: A Virtual Machine Product Line**	Michael Haupt, Stefan Marr, Robert Hirschfeld
47	978-3-86956-130-1	**State Propagation in Abstracted Business Processes**	Sergey Smirnov, Armin Zamani Farahani, Mathias Weske
46	978-3-86956-129-5	**Proceedings of the 5th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
45	978-3-86956-128-8	**Survey on Healthcare IT systems: Standards, Regulations and Security**	Christian Neuhaus, Andreas Polze, Mohammad M. R. Chowdhuryy
44	978-3-86956-113-4	**Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht**	Christoph Meinel, Christian Willems, Sebastian Roschke, Maxim Schnjakin
43	978-3-86956-110-3	**SOA-Security 2010 : Symposium für Sicherheit in Service-orientierten Architekturen ; 28. / 29. Oktober 2010 am Hasso-Plattner-Institut**	Christoph Meinel, Ivonne Thomas, Robert Warschofsky et al.
42	978-3-86956-114-1	**Proceedings of the Fall 2010 Future SOC Lab Day**	Hrsg. von Christoph Meinel, Andreas Polze, Alexander Zeier et al.

ISBN 978-3-86956-193-6
ISSN 1613-5652