Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Declarative Layer Composition with the JCop Programming Language

Malte Appeltauer[a] Robert Hirschfeld[b] Jens Lincke[b]

a. SAP Innovation Center, Potsdam, Germany

b. Software Architecture Group, Hasso-Plattner-Institute, University of Potsdam, Germany

Abstract Program behavior that relies on contextual information, such as physical location or network accessibility, is common in today's applications, yet its representation at the source code level is not sufficiently supported by programming languages. With context-oriented programming (COP), context-dependent behavior can be explicitly modularized and dynamically activated. The available COP implementations offer language constructs that allow to describe context-dependent functionality and to specify for which control flows this functionality should be executed. Since these language constructs require modifications to the source code, the contemporary approaches limit the use of COP to program parts whose source code is accessible to the developer (the user code). The dynamic control over context-dependent behavior in frameworks cannot be directly addressed by COP as this would imply changes to the source code. Instead, context-dependent behavior is addressed whenever a control flow from the framework code enters the user code. Context composition must be addressed at any of these control flow entry points, which may lead to a redundant specification of this functionality. As a result, dynamic control over layers emerges as a crosscutting concern that obstructs the separation of concerns.

In this article, we discuss crosscutting layer composition in framework-based applications in detail. Moreover, we discuss limitations for the expression of semantic relationships of layers that might lead to code duplication. We present a framework-based application, a simple action adventure game that we implemented using a conventional COP language. Along this example, we show how our *JCop* language supports the declaration of layer composition and expression of layer relationships.

Keywords context-oriented programming, dynamic adaption, Java, framework

Malte Appeltauer, Robert Hirschfeld, Jens Lincke. *Declarative Layer Composition with the JCop Programming Language*. In Journal of Object Technology, vol. 12, 2013, pages 4:1–37.
doi:10.5381/jot.2013.12.2.a4

1 Introduction

Modern application design often requires the representation of behavior that strongly depends on its *execution context* [ADB+99, Dey01]. For example, many applications designed for mobile platforms depend on physical location, user activity, or the presence of other users. Such applications need to consider context information for computation in order to better meet the users' expectations. Also, some traditional desktop applications need to take context information into account. For example, some text processing applications dynamically change both their look-and-feel and behavior depending on the users' activity to provide a more intuitive user interaction. Context-specific behavior is not always as perceptible for end users as in the previous examples. More technical examples include switching network protocols depending on the physical environment, or activating an encryption protocol based on the dynamic evaluation of security policies.

We call the execution of a software module whose behavior and response varies depending on contextual information a context-specific *behavioral variation*. For example, a mobile phone may switch its incoming call mode from ringtone to vibrate if the user enters a library, or the copy and paste semantics of a text processing tool may vary if the user switches between plain text and an embedded spreadsheet.

The implementation of behavioral variations requires the modularization of the context-specific behavior, and the dynamic control over its activation and execution. Since a behavioral variation often affects several modules throughout an application, we call it a *crosscutting concern* [KLM+97].

We distinguish between two kinds of crosscutting concerns. *Homogeneous crosscutting concerns* [ALS08] require the same source code to be executed at their *join points* (points in the program's structure or control flow [KLM+97]. A well known approach to the modularization of such concerns is *aspect-oriented programming* (AOP). *Heterogeneous crosscutting concerns* [ALS08] require different source code to be executed at each of their join points. The implementation of behavioral variations is mostly a heterogeneous crosscutting concern, requiring different objects of a control flow to vary their behavior.

The *context-oriented programming* (COP) [HCN08, CH05] approach addresses the representation of such heterogeneous crosscutting concerns. COP proposes the implementation of crosscutting functionality by *partial method declarations* that are able to adapt any common method to their new behavior. Partial method declarations are encapsulated by *layer declarations*. Hence, a layer is the implementation of a behavioral variation. At run-time, the behavioral variations can be composed with the core behavior of the classes by explicitly activating (or deactivating) layers for specific control flows[1].

COP has been applied to several application domains where it showed to be a promising approach for the encapsulation of heterogeneous crosscutting concerns. However, research so far did not explicitly address the incorporation of COP's control-flow based adaptations with application domains that employ *frameworks* [JF88]. For such programs, we distinguish between *framework code*, which is part of the framework implementation, and *user code*, which is part of the concrete application implementation. One property that distinguishes frameworks from libraries is that they prohibit access to their implementation [JF88]. That fact may complicate the declaration of

[1] A few COP implementations do not consider control flow-based layer activation but offer global layer activation instead. In the following, we focus on control flow-based layer activation.

partial methods for base methods inside frameworks, since information about their implementation might be important for the implementation of their partial methods. Still, it is technically possible to declare partial methods for framework methods since the layer declaration is an external module and not part of the framework code.

A problem may occur, if a layer composition must be executed within the framework code because its implementation would require direct modifications to the code, which is prohibited. But even if the framework source code is accessible and could be adapted, the identification of the correct adaptation location would require deep knowledge about the internals and control flow logic of the framework. Obtaining this technical knowledge, in turn, distracts the application programmer from his primary task, i.e., developing the user code.

As a result, the framework code cannot be adapted by layer composition statements. The solution is to move the composition statement instead to a later point during the execution of the control flow at which the control flow enters the user code is executed. This has two advantages. First, the user code is actually accessible and adaptable. Second, the developer should be familiar with this user code and able to implement the adaptation straightforwardly. Unfortunately, control flows that are initiated by framework code often have multiple entry points into the user code. Therefore, the layer composition statements must be repeated at multiple points. That imposes a novel crosscutting concern to our application, which is not driven by the application logic itself but by the layer composition logic. This crosscutting layer composition is a homogeneous crosscutting concern, which requires the same `with` statement to be repeated at multiple points.

In previous publications, we motivated modularization issues of layer composition in COP languages [AHM09] and proposed additional composition features, such as the pointcut-based [AHM$^+$10, AH12], static [SAH11], and reflective [RAL$^+$11] layer composition. We further developed these approaches and integrated them into our Java-based language *JCop*. In this paper, we give a coherent presentation of JCop's composition features. In addition, we discuss its layer inheritance mechanism that eases layer modularization. Throughout the paper, we present JCop's language constructs along a running example of a graphical computer game. Section 2 introduces the core concepts of COP and discusses its issues along our example application. Section 3 presents JCop's declarative layer composition. Section 4 explains language features to express relationships between layers. Section 5 reports on the evaluation of our concepts, which is based on several case studies. Section 6 discusses related work and Section 7 concludes the paper.

2 Context-oriented Programming by Example

In this section we motivate COP along with an example application to which we refer throughout the paper (Subsection 2.1), briefly present our context-oriented Java extension *JCop* (Subsection 2.2), and explain issues concerning the expressibility of the current COP language constructs (Subsection 2.3).

2.1 Example: Crosscutting Behavioral Variations in a Computer Game

In our graphical computer game *RetroAdventure*, the user controls a hero character that moves through a world, speaks to computer-controlled characters, and collects items that are distributed all over the world. The core application is implemented

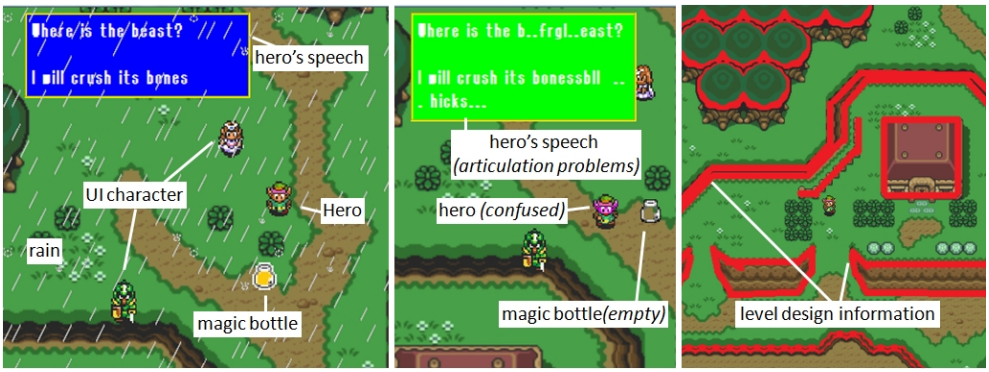


Figure 1 – Screenshots of the RetroAdventure world. *left:* The hero speaks on a rainy day. *middle:* The hero is poisoned by a magic bottle. *right:* Level designer mode with map information. Here: red lines mark the collision area.

in Java and employs the *Swing* [Gal06] GUI framework. A simplified object-oriented decomposition of the user code is shown in Figure 2. In addition to that base functionality, we identified several context-specific concerns that crosscut our class decomposition:

Context-specific character behavior One of the items the hero can collect is a magic bottle. If the hero collects (i.e., drinks) the bottle, he becomes dizzy and confused for several minutes until the bottle magically fills up again. During that time, he cannot properly walk and speak, and the color of his face turns pink.

Environmental conditions The world in which the hero lives can change its environmental conditions, such as temperature, humidity, or daylight. These changes affect pace and accuracy of the characters' movement, their optical range, and responses in conversations.

Level designer mode Besides the context-specific functionality that is directly concerned with the game play, RetroAdventure comes with a debug mode that reveals useful information to a level designer, such as the hero's location coordinates, his movement direction and speed, and a collision map that specifies in which areas the hero is allowed to walk.

Graphics zooming For a better overview of the game world, the user can employ a zooming feature, which implements a level-of-detail approach. It scales the graphics and reloads new images with higher or lower level of detail.

Figure 1 shows screenshots of the application[2]. The implementation of each of these feature crosscuts several classes of our decomposition. These adaptation points are represented by colored circles in Figure 2. For example, the confused hero behavior crosscuts the classes `Hero`, `Character`, `EntityUI`, and `ImageProvider`. In the following, we describe how we implemented such crosscutting concerns using COP.

[2]The pixel art is borrowed from by Nintendo Co., Ltd.

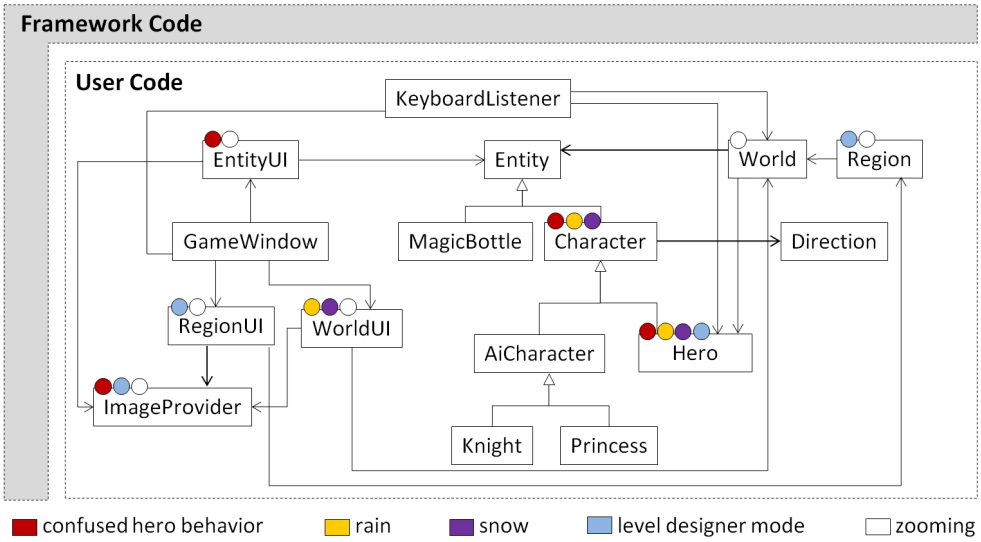


Figure 2 – Class Diagram of the user code of RetroAdventure, where the colored circles represent crosscutting behavior.

2.2 Context-oriented Programming with JCop

COP allows the modularization of crosscutting concerns by *layer declarations*. 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. Listing 1 presents examples of these terms in JCop syntax[3].

Layer Declaration To distinguish a common (Java) method from methods whose behavior is adapted by layers, we call the former *plain method declarations* (Line 2). *Layered method declarations* are the counterpart to plain methods and describe methods that are potentially adapted by at least one layer (i.e, whose behavior may be modified by a layer at run-time). A layered method declaration consists of a *base method* and at least one *partial method*. The term refers to the static declaration of plain and partial methods, which is independent of the layer activation at run-time[4].

Base method declarations are the 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 (Lines 3–4)[5]. *Partial method declarations* are declared within a layer and implement a behavior variation of a base method (Lines 7, 12–14, 18–20)[6]. *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 (Lines 15). The enclosing class of a base method is called the *host class* of the base method and its partial method (Lines 1–9).

[3]JCop can be downloaded at https://www.hpi.uni-potsdam.de/swa/trac/Cop/wiki/JCop.
[4]In terms of AOP, layered method declarations, in particular their partial method declarations, describe the static join point shadows of the layers' behavioral variations.
[5]The base method declarations can be regarded as the static join point shadows of layers.
[6]A partial method declaration can be compared to an advice body in AOP.

```
1  public class Hero {
2    public Point getPosition() {...} //plain meth.
3    public void move(Direction dir) {...} //base meth. for layered method Hero.move
4    private Position getPos() {...} //base meth. for layered method Hero.getPos
5
6    layer ConfusedHeroLayer { //layer extension declaration
7      private Position getPos() {...} //partial meth. for layered method Hero.getPos
8    }
9  }
10
11 public layer ConfusedHeroLayer { //(top-level) layer declaration
12   public void Hero.move(Direction dir) { //partial meth. for layerd meth. Hero.move
13     proceed(dir); //a proceed expression
14   }
15   private Direction getNewDir(Direction original) {...} //layer local method
16 }
17 public layer Rain { //(top-level) layer declaration
18   public void Hero.move(Direction dir) { //partial meth. for layerd meth. Hero.move
19     proceed(dir); //proceed expression
20   }
21 }
```

Listing 1 – A layer and partial method declaration in JCop.

In JCop, layers are type declarations that can contain *partial method declarations* and the standard class member declarations (i.e., classes, methods, enums, and fields). 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, layer declarations can be extended (i.e., *opened*) in any class to add partial member declarations (Lines 6–8).

Partial method declarations qualify their base method 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.

The implementation of behavioral variations can be regarded from two perspectives. A developer may focus either 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, partial methods should be implemented within the layer declaration. Thus, the interaction (and coupling) among the partial methods can be implemented easier, since the partial methods are defined in the same scope. For example, the partial methods of an HTML rendering layer that visits a document structure would probably contain similar functionality (i.e, the rendering of the different document nodes) and require the same auxiliary methods for pretty printing. Therefore, it makes sense to declare these methods physically close to each other (i.e., in the same layer declaration). Contrary, if the focus is on individuality and close iteration with their host class, the partial methods should be implemented physically close to their base method definition (i.e, in the host class) rather than in their top-level layer declaration. For example, in a editor for structured text, a layer may add a text formatting action in a tool tip. The the partial methods implement quite different functionality, such as the rendering of widgets, parsing the document tree, and applying formatting rules. These partial methods have more dependencies to their respective host classes than among each other. Therefore, we would implement them physically close to their base functionality (i.e., in the same host class).

As a result, JCop's layer definitions can be extended within classes with additional

partial methods. Layer declaration extensions can only contain partial methods of base methods that are defined or inherited by the enclosing class. This mechanism also allows adaptation of private or protected methods in Java with respect to their visibility. In Listing 1, we chose to implement the partial method `move` within the layers (Lines 12–14, 18–20). Alternatively, we could extend the layer declaration within the class[7].

Layer Composition Layers can be composed with the base system per control flow. To express layer composition, JCop provides the `with` block statement (containing an argument list for the layers to be activated) that can be used in the bodies of methods, constructors, and initializers. The specified layers are only active for the *dynamic extent* of the `with` 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. The following code shows an activation of `ConfusedHeroLayer` and `RainLayer` and the method invocation of `Hero.move` within the block of a `with` statement:

```
public void moveHeroLeft() {
  with (new ConfusedHeroLayer(), new RainLayer()) { getHero().move(Direction.LEFT); }
}
```

To explicitly disable layer execution for a certain control flow, JCop offers the `without` (for a specific layer instance) and `withoutall` (for any instance of a specific layer type) block constructs.

The `proceed` Expression A layer composition can contain several partial methods of a layered method. On method invocation, the call is delegated to the partial method of the outermost layer. To explicitly invoke the next partial method declaration, we can use the pseudo-method `proceed` (Listing 1, Lines 13, 19). The order of the partial methods that can be traversed by `proceed` is determined by the layer activation order. In the listing above, `ConfusedHeroLayer` is activated before `RainLayer`. Therefore, the call to `move` is first send to `ConfusedHeroLayer`. If its partial method contains `proceed`, it is delegated to `RainLayer`. In turn, if this partial method contains `proceed`, the base method is called. Note that the base method is always the last method in the composition chain. If one partial method in the composition chain does not contain `proceed`, the succeeding partial methods and the base method are not executed. There is no way to bypass the layer composition chain and directly call the base method. Both the return type and the expected arguments of `proceed` must conform to the method's signature.

2.3 Problems

In the previous section, we introduced the language constructs (layer declaration and explicit layer activation) that are provided by all (layer-based) COP languages we know of. This layer-based implementation allows for an explicit representation of context-specific behavior by encapsulation of crosscutting concerns and run-time control over their activation.

[7] In literature, partial method declarations within top-level layers are also called *class-in-layer* declarations. Layer extension declarations are an instance of *layer-in-class* declarations [CH05].

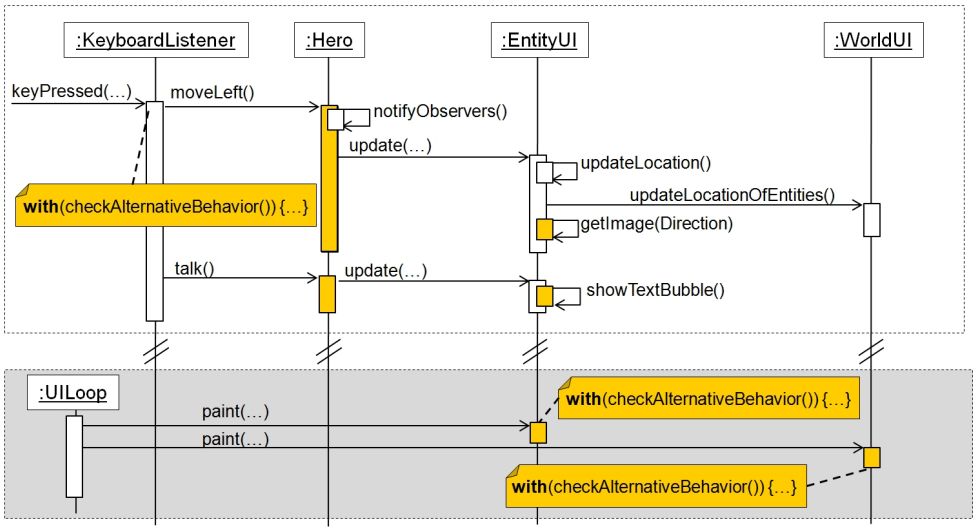


Figure 3 – Redundant layer compositions in RetroAdventure (*gray:* framework thread, *white:* main thread).

However, the use of these language features imposes new modularization issues, namely *crosscutting layer compositions* and *inexpressible relations between layers*, which are explained in the following.

2.3.1 Crosscutting Layer Compositions

In our implementation of RetroAdventure, we encountered some issues concerning the specification of the dynamic layer composition that demand for more flexible composition expressions.

We explain that issue by the example of the layer `ConfusedHero` that implements the context-specific character behavior. This layer should be activated whenever the magic bottle is empty (i.e., whenever the hero recently drunk the bottle). The corresponding layer activation is implemented by the following `with` statement:

```
with(currentHeroBehavior()) { ... }
```

and the following auxiliary method[8]:

```
Layer currentHeroBehavior() {
  if(getBottle().isEmpty()) return new ConfusedHeroLayer();
  else return null;
}
```

Figure 3 (white box) presents a sequence diagram of a user interaction. The `with` statement is used in the `keyPressed` method of the `KeyboardListener` class, so that any user-triggered control flow can activate the layer. So far, we are able to concisely express the composition of `ConfusedHero` at only one point in our class decomposition – namely within the keyboard listener callback method.

However, during the execution of this user-triggered control flow, other threads may asynchronously call methods that are layered by `ConfusedHero` but without having

[8]We assume that the method `Bottle.isEmpty` returns `true` for a period of time after being drunk by the hero and then switches back to false meaning that the bottle has been magically filled up again.

```
import model.*;
import ui.*;

public layer Rain{
 public void WorldUI.renderMap(){
  // render rain
 }
 public void Character.move(Direction dir){
  // calculate the new pace and direction
 }
 public void Character.talkTo(Character c){
  // mix *atishoo* into the conversation
 }
}
```

```
import model.*;
import ui.*;

public layer Snow{
 public void WorldUI.renderMap(){
  // render snow
 }
 public void Character.move(Direction dir){
  // calculate the new pace and direction
 }
 public void Character.talkTo(Character c){
  // mix *brr* into the conversation
 }
}
```

Listing 2 – Two semantically similar layers that require redundant partial method declarations.

this layer activated. For example, the Swing framework may asynchronously call the `paint` methods of the classes `EntityUI` and `WorldUI`, which both provide partial methods for `paint`. Because these framework-triggered control flows do not pass the `keyPressed` method (and its `with` statement), they only execute the base declarations of the `paint` methods and ignore their partial declarations. Therefore, the control flows of the UI thread must be extended with a layer composition as well. Figure 3 (gray box) shows this framework-triggered control flow. Because we cannot access the source code of the UI loop[9] inside the framework, layer composition is moved to the entry points of the framework-triggered control flows into the user code. As an effect, the layer composition statements are redundantly implemented at several source code locations.

With that, layer composition is now a crosscutting concern in our implementation. Obviously, this fact contradicts the intention of COP to support the separation of concerns. Section 3 describes how crosscutting layer composition can be modularized using context classes in JCop.

2.3.2 Inexpressible Relationships between Layers

The behavior of layers can be semantically related. In the RetroAdventure world, it can either rain or snow, but not both, see Listing 2. Therefore, it should be possible to declare the two layers to be mutually exclusive.

Moreover, both layers `Rain` and `Snow` adapt the base program at the same base methods. As they provide similar functionality, they also require similar subroutines that must be redundantly declared for each layer, see Listing 2. Such relationship between layers cannot be directly expressed by means of COP implementations so far. In Section 4, we describe how JCop addresses this issue.

3 Declarative Expression of Crosscutting Layer Compositions

In the previous section, we discussed some of the limitations of current COP implementations: If applied to specific application architectures, it may be impossible to concisely declare a layer composition. Instead, adaptation statements are scattered over the application, which in turn hinders good separation of these crosscutting concerns. JCop offers means to declaratively express layer compositions. Layer com-

[9] For simplification, we represent the main UI thread loop by a class `UILoop` in the diagram.

```
1  public contextclass MagicBottleContext {
2    private MagicBottle bottle;
3    private Layer confusedBehavior = new ConfusedHeroLayer();
4
5    public MagicBottleContext(MagicBottle bottle) {
6      this.bottle = bottle;
7    }
8    public boolean heroDrunkTheBottle() {
9      return bottle.isEmpty();
10   }
11   when(heroDrunkTheBottle()) : with(confusedBehavior);
12 }
```

Listing 3 – A context class handling the composition of `ConfusedHeroBehavior`.

positions can be expressed by *composition predicates* that are encapsulated in a *context class* module (Subsection 3.1). Furthermore, layer declarations can be declared globally active (Subsection 3.2).

3.1 Context Classes

JCop's declarative layer composition is implemented by a domain-specific aspect oriented language [KLM+97]. Its join point model consists solely of method executions, which 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. The advice composition contains a sequence of `with` and/or a `without` operators. Layer composition pointcuts can be composed using the logic operators disjunction, conjunction, and negation. 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, similar to an `execution` pointcut in AspectJ-like languages [KHH+01, AGMO06][10].

The `when` pointcut allows for a more implicit description of layer composition, 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 `MagicBottle` object) that can be evaluated by a boolean expression. Whenever this expression evaluates to true, the layer composition is applied. Listing 3 presents a context class that declares a `when` pointcut to evaluate whether the hero should apply its confused behavior (Line 11).

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, the pointcut expressions are considered to be side-effect free[11]. In order to guarantee that a dynamic extent is executed with a consistent layer composition, we impose an additional restriction to the `when` evaluation. JCop ensures that, once a `when` pointcut is evaluated, it is not re-evaluated within the dynamic extent

[10] In the following, we assume that `on` pointcuts use this signature-based join point specification. However, using an annotation-based approach as used in AspectJ, would be also possible conceptually.

[11] A similar restriction is specified for expressions used in guard predicates in the ObjectTeams language [HHM07].

originating from this evaluation. 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 desired, the `when` pointcut can be labeled with an annotation. The annotation `@CheckOncePerCflow` represents the default behavior explained in the previous paragraph (i.e., no annotation is specified). The annotation `@CheckAlways` denotes a continuous evaluation at any method execution. This may be useful if the expression to be evaluated has side effects that should be always executed.

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.

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. As a type 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 pass variables to their advice block.

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 applied to the composition in the order they occur in the advice declaration. The following advice activates first an instance of `ConfusedHeroLayer` and `ZoomingLayer`, and then deactivates all active layers of type `RainLayer`:

```
on(...): with(new ConfusedHeroLayer(), new ZoomingLayer()), withoutall(RainLayer.class)
```

This corresponds to the following explicit declaration, which would be required at any matching join point:

```
with(new ConfusedHeroLayer()) {
  with (new ZoomingLayer()) {
    withoutall(RainLayer.class) {
      ...
} } }
```

Context Class Declaration Declarative compositions and named pointcuts 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`.

3.2 Static Active Compositions

Layers Layers can also be used for static extensions of the application, which is denoted by the use of the `staticactive` modifier in the layer declarations. At the initialization of a program, a singleton instance of all static active layer is created and added to the composition. For the initialization of the singleton, the default

```
1  void initMagicBottles() {
2    MagicBottle greenBottle = new MagicBottle();
3    MagicBottle redBottle = new MagicBottle();
4    MagicBottleContext greenBottleCtx = new MagicBottleContext(greenBottle);
5    MagicBottleContext greenBottleCtx = new MagicBottleContext(redBottle);
6
7    greenBottleCtx.deploy(); // composition: [eval(greenBottleCtx), base]
8    with (new RainLayer() ) { // composition: [RainLayer, eval(greenBottleCtx), base]
9      redBottleCtx.deploy(); // composition: [eval(redBottleCtx), RainLayer,
10     ... // eval(greenBottleCtx), base]
11   } // composition: [eval(redBottleCtx), eval(greenBottleCtx), base]
12   redBottleCtx.undeploy(); // composition: [eval(greenBottleCtx), base]
13 }
```

Listing 4 – Instantiation and activation of two context classes.

constructor of the layer is used. This feature simplifies the declaration of crosscutting concerns that should be active during the entire execution of an application. At run-time, an introspection of the composition list (using `Composition.current()`, see Appendix) returns at least all static active layers and the base method. To load and activate static active layers, the JCop run-time needs to be parametrized with their full qualified names. For that, the JCop compiler offers the option *-staticactive <typenames>*. The composition order of static active layers is defined by the order of the parametrization. Likewise, static active layers can be passed to the JCop launcher using the same option.

In RetroAdventure, we consider the zooming feature as a crosscutting concern that should statically extend our application, similar to an class extension. Therefore, we implement this functionality as a static active layer. The following line compiles RetroAdventure and uses the `ZoomingLayer`:

```
jcopc -sourcepath src -d bin -staticactive gui.zooming.ZoomingLayer main.RetroHeroInit
```

Context Classes By default, context classes must be initialized and deployed in the control flow (see Subsection 3.3). In addition, the modifier `staticactive` declares that one *singleton* instance of the context class is implicitly globally active. The advice of these static contexts potentially affect every running thread in the system. Note that there is only one instance of a static active context class. In terms of AOP, a static active context class declaration is comparable to an default singleton aspect declaration in AspectJ [KHH+01]. With that, a context class is not connected with particular code fragments or control flows within the code but, instead, serves as a global context-dependent behavioral adaption affecting the whole application. To be exposed to the compiler and launcher, static active context classes use the same mechanism than static active layers.

3.3 Execution Order at Shared Join Points

Declarative layer compositions are evaluated at context class instance level. Hence, multiple instances of a context class can be active in the same control flow. Context class instances can be deployed and undeployed, in a similar way to dynamic aspects languages, such as CaesarJ [AGMO06]. In order to activate the advice of a context class instance, the method `deploy` must be called. Likewise, the method `undeploy` disables the advice execution of a context class instance. Listing 4 shows an example

for the initialization of two magic bottles, their binding to a context class instance, and the activation of the context classes.

It is possible that composition advice of multiple instances — either of the same or of different context classes — share the same pointcut. In this case we need a well defined order of composition advice execution.

In COP, the partial method execution order is controlled by the programmer through the order of explicit `with`-statements in the code. The same assumption holds for context classes. Within a context class advice body, the sequence order of `with` and `without` expressions determines the order in which the layers are applied to the composition, see Subsection 3.1.

The order in which the layer compositions of two distinct context classes are applied is determined by the deploy order of the instances, which is expressed using the `un-/deploy` methods that context classes inherit from their implicit super type `jcop.lang.ContextClass`, see Appendix. The last deployed context class instance is called before the other context classes. In our example in Listing 4, the advice of the red bottle context is executed first since it is deployed first (Line 7). On the execution of a layered method, the layer-aware dispatch first evaluates the advice of `greenBottleCtx` and adds its layers to the composition. Then, it dispatches to the base method. The next line of the listing contains an explicit with statement, which is added to the composition list and would be the first to call. Next, we deploy `redBottleCtx`, which, in turn, is moved to the end of the composition. Now, the layer composition at least contains the rain layer and the base layer. Depending on the result of the evaluation (the join point matching), `redBottleCtx` provides layers that are placed before the rain layer, and `greenBottleCtx` provides layers between the rain layer and the base method. In Line 11, the block of the explicit with statement ends. Thus, the rain layer is removed from the composition, while the two context classes are still contained.

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 adds composition statements for future computations to an ordered list.

4 Expression of Layer Relationships

In RetroAdventure, some layers provide alternative behavior, are mutually dependent, or exclusive. Similar to alternative subclasses, alternative layer declarations often share parts of their state or behavior. In such cases, it is convenient if the programming language provides two features. First, a means to minimize code repetition by code reuse and second, a means to represent relationships between layers. Since these relationships may be dynamic, a static declaration of them would not be flexible enough. JCop addresses the former by class-based inheritance for layer types (Subsection 4.1) and the latter by reflective means that allow to programmatically specify declare layer relations (Subsection 4.2). For applications that require special reasoning about layers and their activation, the reflection API provides access to the required information (Subsection 4.3).

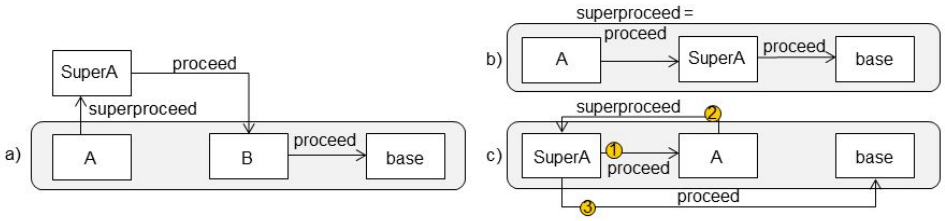


Figure 4 – Examples for `proceed` and `superproceed` in layer compositions (*gray boxes*).

4.1 Class-based Inheritance

JCop's layer type declarations come with a class-based layer inheritance mechanism. The implicit super type of a layer is `jcop.lang.Layer`, see Appendix B.1. Optionally, layers can declare a super layer using Java's `extends` declaration. Layers may only inherit from other layers. They cannot inherit from classes.

Scoping The introduction of super layers requires an extension to the scoping mechanisms known from Java. The scope of a partial method declaration in a layer type is the method's host class. Contrary, the scope of a class member declaration in, or inherited by a layer type is the entire layer body, including any nested type declarations.

As a consequence, the behavior of Java's keywords `this` and `super` must be specified for their use within partial method definitions. In partial methods, the `this` keyword refers to the host object and the `super` refers to host object's super type. For the explicit access to 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.

Member references without an explicit receiver object or a `this`, `thislayer`, `super`, or `superlayer` keyword are first looked up in the enclosing layer, then in the scope of the target object. This lookup order is based on our experience that partial methods access layer local methods more often than members of their host class.

Partial Method Overriding A partial method m_A declared in a layer `A` overrides a partial method, m_{SuperA}, declared in layer `SuperA` if `A` is a sublayer of `SuperA` and the signature of m_{SuperA} is *equal* to the signature of m_A, including all partial method modifiers (except `abstract`[12]). Layers can override partial methods of their super types. 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.m()`[13]. To support super access for partial methods, JCop provides a variation of the `proceed` expression, called `superproceed`.

While `proceed` calls the next corresponding partial method in a layer composition chain, `superproceed` calls the partial method declaration of a super layer. The super layer does not necessarily have to be an element of the composition chain. Figure 4 (a) shows a layer composition where a layer `A` uses `superproceed` to call its superlayer. Note

[12] `Super` and m_{SuperA} may be declared abstract and implemented by m_A.

[13] Otherwise, the same mechanism would allow explicit calls of any partial method of a super layer, with unclear semantics.

```
public abstract layer Weather {
  // abstract partial methods, define variation points
  abstract public void WorldUI.renderMap();
  abstract public void Character.talkTo(Character c);
  abstract public void Character.move(Direction dir);
  // layer local method
  public Direction slowDown(Direction dir, int speed) { ...}
}

public layer Rain extends Weather {
  public Composition onWith(Composition current) {
    return current.withoutAllLayer(Snow.class);
  }
  // concrete partial methods, implement variation points
  public void WorldUI.renderMap() {...}
  public void Character.talkTo(Character c) {...}
  public void Character.move(Direction dir) {slowDown(dir, 5);}
}

public layer Snow extends Weather {
  public Composition onWith(Composition current) {
    return current.withoutAllLayer(Rain.class);
  }
  // concrete partial methods, implement variation points
  public void WorldUI.renderMap() {...}
  public void Character.talkTo(Character c) {...}
  public void Character.move(Direction dir) {slowDown(dir, 10);}
}
```

Listing 5 – Layer inheritance and implicit layer-based activation.

that in this case the super layer is not part of the composition. A `proceed` within the superlayer will call the next layer in the composition, which is `B`.

If both the layer and its super layer are elements of the composition chain, they are treated like two distinguished layers and executed in the order of their activation. If the sublayer is called before the superlayer, `proceed` and `superproceed` have the same behavior, see Figure 4 (b). If the superlayer is called before the sublayer, `superproceed` calls the superlayer a second time. Note that lookup cycles cannot occur, since `proceed` is evaluated dynamically. A `proceed` within the sublayer calls the next layer in the composition, which is the base layer, see Figure 4 (c).

The declaration of `superproceed` is statically checked. Therefore, if no super layer implements a corresponding partial declaration its use causes a compile-time error.

Abstract Layers Layer declarations can be declared abstract. In combination with partial method declarations, this feature can be used to separate the specification of adaptation points from their implementation. Abstract layers may contain concrete members, similar to abstract classes. Typically, the identification of adaptation points in a large systems requires deep domain knowledge. Once these adaptation points are identified, the implementation of behavioral variations may be more independent of the system internals and therefore easier to implement by developers with less experience of these internals. This use of abstract layers is similar to the use of abstract aspects in AspectJ [KHH+01].

Listing 5 illustrates how we separate identification and implementation of adaptation points using layer hierarchies. The example shows the layers `Rain` and `Snow` that adapt the system at the same layered methods. Because of this similarity, we introduce an abstract layer as a common superlayer of the two layers. The abstract layer `Weather` implements some auxiliary methods that can be used by its concrete sublayers.

```
public layer Rain extends Weather {
  public Composition onWith(Composition current) {
    if (World.finalQuestRunning()) {
      if (!current.contains(Snow.class))
        return current.withoutLayers(new Snow());
    }
    else
      return current.withoutAllLayer(Snow.class);
  }
}
```

Listing 6 – Different implicit layer activation strategies.

In addition, this abstract layer defines abstract partial methods to be implemented by concrete weather layers. The sublayers `Rain` and `Snow` can be implemented without identifying these adaptation points. Instead, the compiler tells the developer which partial methods have to be implemented. This way, we can use abstract layers to declare a simple crosscutting interface [GSS+06].

4.2 Composition Event Handlers

Some application-specific dependencies of layers may also impact their composition. For example, in RetroAdventure, the layer `Rain` must not be composed with the layer `Snow` (sleet does not exist in our fantasy world). JCop allows the expression of such layer-based composition rules through 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 (see Appendix B). The handlers are called for explicit and declarative layer composition (`with` statements and context classes). However, they are not called for reflective layer activation (see Subsection 4.3), which could lead to infinite loops since it could be used within the event handlers themselves. The handlers are called 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.

Listing 5 shows the implementation of such rules for the two layers `Rain` and `Snow`. To assure that instances of both layers are never active at the same time, we override `onWith`. Here, we could either throw a `JCopCompositionExeception` that indicates the illegal layer composition. In our example however, we decide to solve the conflict by mutual deactivation. On activation of each of the two layers, its composition handler method takes care that any instance of the other one is removed from the list.

For the special case of mutual exclusion, one could also think about a rule-like syntax extension that is more explicit than our above implementation using composition event handlers. The EventCJ language (see Subsection 6.1.1) provides such a rule engine to switch between layers. Our approach is more flexible since it evaluates the rules at run-time and also takes dynamic changes to relations into account. If, for example, some quests our RetroAdventure hero has to solve allow for more dramatic weather conditions, we could extend our rules by run-time checks to change the dependency. Listing 6 presents an alternative behavior of the rain layer composition handler if the final quest of RetroAdventureis running. It checks if a snow layer is

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

Listing 7 – Using reflection to instrument new threads with a layer composition.

part of the composition. If not, it activates a new layer instance. The implementation of the snow layer handler method is similar.

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.

The complete reflection API is documented in the Appendix B. Its class `Composition` contains an ordered list of composed layers. It provides access and navigation through the composition's layers. The methods `withLayers`, `withoutLayers`, and `withoutAllLayers` correspond to the composition statements `with`, `without`, and `withoutall` 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. Listing 7 presents the usage of the reflection API. In JCop, a new thread does not inherit the composition of its creating thread. However, using reflection we can easily initialize the new thread with the old layer composition and also add new layers to the new composition. 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 `Rain` is generated. An array containing the composition's layers is then used as an argument of the `with` statement.

5 Evaluation

We observed the issue of crosscutting layer compositions not only in the RetroAdventure case study, but also in several other case studies that we conducted and presented in previous work. In this section, we give a brief overview of these projects (Subsection 5.1) and describe the results of our refactoring to JCop (Subsection 5.2).

5.1 Case Studies

CJEdit [AHM+10, AHHM11] is a simple graphical source code editor that provides two modes of user operation: *programming* and *documenting*. The programming mode is supported by syntax highlighting, an outline view, and a compilation/execution toolbar. The documenting mode allows formatting Java compilation units with rich text comments. Both operation modes, require separate graphical elements (menus, views, and toolbars) and functionality that require changes at several parts of the application. Moreover, the switch between the operation modes depends on

```
public void onCursorPositionChanged() {
  if (blockTypeChanged()) {
    with (getLayersOfCurrentBlock()) { drawWidgets(); }
} }
public void onSave() {
  if (blockTypeChanged()) {
    with (getLayersOfCurrentBlock()) { saveDocument(); }
} }
public void onPrint() {
  if (blockTypeChanged()) {
    with (getLayersOfCurrentBlock()) { printDocument(); }
} }
public void onFileNew() {
  if (blockTypeChanged()) {
    with (getLayersOfCurrentBlock()) { createDocument(); }
} }
```

Listing 8 – Redundant layer composition in CJEdit's event handlers.

dynamic properties. Therefore, we decided to implement the operation modes by layers and the switch between them by layer composition blocks.

The text editor's core is implemented using Java and the *Qt Jambi GUI Framework* [NC09]. It consists of 33 classes (approximately 3500 lines of code), where most parts are written using plain Java constructs. The implementation of the user operation modes consists of 6 layers (approximately 330 lines of code).

Some characteristics of GUI-based programming led to additional challenges for the layer implementation similar to what we experimented during the development of RetroAdventure. User interaction with GUI behavior is event-driven rather than activated by control flow. The main event-loop is hidden in the framework's internal implementation, which is inaccessible for the adaptation by `with` statements. Hence, we have to declare composition statements redundantly within our application source code instead of within the framework's event loop.

Listing 8 presents the event handler methods and their redundand with statements.

We addressed that problem by context classes. CJEdit also contains layers with overlapping functionality, as shown in Listing 9[14]. Both `RTFWidgets` and `CodeWidgets` share extend the same layered methods and require similar auxiliary methods. To get rid of this redundant declaration, we used layer inheritance for their JCop-based implementation.

WhenToDo [RAL+11] is a mobile todo application that helps to prioritize tasks depending on the current working environment and situation. For example, specific tasks require Internet access, or can only be accomplished at a specific location. If the required resources are available, the application reminds the user to complete the respective task. Instead of implementing context management and analysis ourselves, we base our application on an existing system [RAL+11, RSC06]. This system allows the specification of context-sources (such as a *NearbyContact* service) and provides a rich context query language. Queries can be executed using the context management interface in two ways: Either, the query is directly evaluated and returns whether the context is active. Or, an observer is registered for each query and notifies the application on any future context change.

[14]Note, that these declarations are layer declaration extensions and declared inside their host class `TextEditor`, which is why they are able to extend private methods.

```
public class TextEditor {
  layer RTFWidgets {
    //partial methods
    after private void drawMenus() { ... }
    after private void drawToolBars() { ...}
    // auxiliary members
    private QMenu formatMenu;
    private FormatToolBar formatToolBar;
    private CodeToolBar createToolBar() {...}
    private Menu createMenu() {...}
  }
```

```
layer CodeWidgets {
  // partial methods
  after private void drawToolBars () {...}
  after private void drawMenus () {...}
  // auxiliary members
  private CodeToolBar codeToolBar;
  private Menu codeMenu;
  private CodeToolBar createToolBar() {...}
  private Menu createMenu() {...}
}
}
```

Listing 9 – Code repetition in CJEdit's layers.

The implementation consists of the actual WhenToDo application which is integrated into a simple mobile device simulator. WhenToDo is implemented by 9 classes (approximately 400 lines of code). The simulator consists of 14 classes (approximately 1000 lines of code)[15].

In this application, layer composition depends on external context change events that may occur anytime. To anticipate this asynchronous context change during program execution, we have to introduce a layer composition block guarded by a condition at each relevant source code location. To avoid these redundancies, we implemented a context class to deal with this crosscutting concern. For example, we use the context class `NearbyContacts` to manage context changes. Context queries are then passed to the context class that handles the query's result and composes new layers when the context changes.

AstroPic [SAH11] is an image gallery application for the mobile device platform *Android* platform [Ope]. AstroPic automatically downloads and displays the current astronomy picture of the day with a short descriptive text. The application is implemented for Android as a simple graphical user interface that asynchronously downloads the current image from the Web. Its download strategy depends on the network availability, for which several layers provide alternatives.

The AstroPic Android application consists of nine classes and two layers for network bandwidth-specific behavior (approximately 320 lines of code).

The application programmer's control over the threads and callbacks in Android is limited, which complicates explicit layer activation and again causes redundant layer composition blocks. Therefore, we employed a context class for a declarative specification of layer composition. However, this does not entirely solve the problem of thread control but shifted it from layer activation to context activation. Now, the application code that may be executed by different threads potentially uses different active context classes and with that different layer compositions. As a solution, we used the `staticactive` declaration of context classes. These static contexts potentially affect every running thread in the system. Thus, the context class is not connected with particular code fragments or control flows within the code but, instead, serves as a global context-dependent behavioral adaption affecting the whole application.

MyBook [AH12] is a simple Web service-based book shop, whose client and services are implemented using COP. It offers a book search that considers user-profile

[15] We omitted the implementation of a GUI for the mobile phone and its applications.

	Case Studies				
	RetroAdventure	**CJEdit**	**WhenToDo**	**AstroPic**	**MyBook**
Project Size (User Code)					
decomposition	33 classes 6 layers	40 classes 6 layers	25 classes 2 layers	9 classes 2 layers	30 classes 6 layers
lines of code	3000	3500	1500	320	4500
Explicit Layer Composition					
implementation	15 *with* blocks	13 *with* blocks	18 *with* blocks	3 *with* blocks	10 *with* blocks
tangled classes	**30%**	**13%**	**24%**	**33%**	**24%**
Context Classes and Static Active Layers					
implementation	2 context classes 1 static active layer	2 context classes 1 layer-based composition	1 context class context query library	1 context class	1 context class JCop/SOA library
tangled classes	**0%**				
Layer Declaration without Inheritance					
overlapping source code	2 layers	2 layers	/	/	2 layers
% duplicated code	**60%**	**85%**	**/**	**/**	**58%**
Layer Declaration with Inheritance					
overlapping source code	0 layers	0 layers	/	/	0 layers
% duplicated code	**0%**	**10%**	**/**	**/**	**5%**

Figure 5 – Overview of the ContextJ/JCop implementations of layer composition in framework-based applications.

information such as age and visual defects. If, for example, a juvenile customer performs a book search, the result is filtered and inappropriate books and advertisements (banners) are not listed; if a customer has visual problems reading the Web page, it is rendered with larger font size and images.

The MyBook Web shop is implemented using *Enterprise Java Beans* and the *JBossWS* Web service framework [Red11], which we extended to attach layer composition information to remote method calls. The frontend consists of 14 classes and 3 layers, the backend contains 12 classes and 3 layers (approximately 4500 lines of code).

5.2 Results

Figure 5 gives an overview of these case studies. We first implemented the applications using our plain COP language *ContextJ* [AHHM11]. ContextJ supports simple layer declaration – not capable of event handlers, global activation, and inheritance – and explicit layer composition – using the `with` block. As the table shows, layer composition is scattered over up to 33% of the user code classes. In three applications exist layers that are semantically related contain up to 85% code duplication.

We then refactored the applications to JCop and used the features described previously. In all case studies, layer composition could be fully encapsulated by context classes and static active layers. Layer inheritance allowed us to reduce almost any code duplication within the layers, so that the modularization of the behavioral variations has been advanced. For example, the layer composition within CJEdit's event handlers (see Listing 8) is now represented by a context class, see Listing 10. The widget layers now inherit the partial methods to be implemented by their new superlayer `UserModeWidget`, see Listing 11.

```
public contextclass UserActivityChange {
  private CJEditWindow win;
  public UserActivityChange(CJEditWindow win) {
    this.win = win;
  }
  pointcut eventHandlerCall :
    on(void CJEditWindow.onPrint()) || on(void CJEditWindow.onSave()) ||
    on(void CJEditWindow.onFileNew()) || on(void CJEditWindow.drawWidgets());
  eventHandlersCall && this(win) : with(win.getLayersOfCurrentBlock());
}
```

Listing 10 – A context class declaring the composition at CJEdit's event handlers.

```
  public abstract layer UserModeWidget {
    protected QMenu formatMenu ;
    protected FormatToolBar formatToolBar ;
    protected CodeToolBar createToolBar() {...}
    protected Menu createMenu() {...}
  }
  public class TextEditor {
    layer UserModeWidget {
      abstract after private void drawToolBars();
      abstract after private void drawMenus();
    }
    layer RTFWidgets { /*implement the abstract partial methods */ }
    layer CodeWidgets { /*implement the abstract partial methods */ }
  }
```

Listing 11 – The abstract layer `UserModeWidget` declares auxiliary methods and abstract pointcuts.

6 Related Work

In this section, we look at programming languages that are related to JCop. We discuss related approaches along JCop's language features for layer composition (Subsection 6.1) and layer declaration (Subsection 6.2). Finally, we present language approaches related to COP (Subsection 6.3).

6.1 Layer Composition

We first discuss the language design and implementation of related COP languages with respect to their layer composition constructs. Figure 6 presents an overview of the COP languages mentioned in this section and summarizes their features and implementation.

6.1.1 Pointcut-based Declaration

The *EventCJ* [KAM11] language is closely related to JCop[16]. Both languages are based on *ContextJ* [AHHM11] and extend it with a domain-specific pointcut language

[16]The first paper mentioning EventCJ [KAM10] was published, shortly after our first publication describing JCop [AHM+10].

for declarative layer activation. However, they use different built-in pointcuts and advice semantics.

EventCJ inherits the entire AspectJ join point model and supports all pointcuts of the AspectJ language. Pointcuts are used to declare events. These events can be handeled by layer transistion rules to manage layer composition. The following listing (taken from [KAM11]) presents the declaration of an event `GPSEvent` and a transition rule that switches the activation of two layers `WifiNavi` and `GPSNavi`.

```
declare event SwitchDevice(Navigation navi):
  after call(void Navigation.onStatusChanged(..)) && target(navi): sendTo(navi);
transition GPSEvent: WifiNavi switchTo GPSNavi
```

Like JCop, EventCJ uses pointcuts to express the join points of a layer activation. However, JCop restricts its join point model to method executions and dynamic conditions. Furthermore, the layer composition within the advice is different in both languages. Declarative layer activation in EventCJ is defined by transition rules that can express conditional layer activation.

EventCJ also provides layer activation handlers that can execute additional functionality on layer composition. The handlers are declared by special keywords, `activate` and `deactivate`:

```
class A { ... layer Alpha { activate{ ... } deactivate{ ... } } }
```

In contrast, JCop layer activation handlers do not require a syntax extension but are implemented as methods. In addition, they can influence the method composition, as described in Section 4.2.

Except for declarative compositions, EventCJ does not support other features that were added to JCop in addition to the ContextJ features. Thus, EventCJ does not support top-level layer declaration, layer hierarchies, and layer instantiation.

6.1.2 Instance-specific and Object Structure-based Layer Composition

The JavaScript extension *ContextJS* [LASH11] addresses the need for additional scoping strategies, such as instance-specific and structural scoping, and proposes an open implementation for COP layer composition. This open implementation allows developers to define domain-specific scoping strategies. With JCop, we cannot directly define such new scopes. ContextJS, in turn, cannot concisely encapsulate scattered composition statements.

6.1.3 Event-based Layer Composition

An important difference between event-based programming and event-based context (de)activation needs to be highlighted. Event-based programming supports the synchronous or asynchronous trigger of an *immediate* action as events are signaled. Conversely, event-based context (de)activation triggers recomposition, which causes the binding of actions at interfaces that may be executed *in the future*. Obviously, layer composition events have a certain influence on action characteristics, but this is expressed only in terms of bindings of actions to interfaces; their actions are not immediate (synchronous or asynchronous) results of events.

The CaesarJ extension *ECaesarJ* [NNG09] supports the definition of context as a class which implements two events that represent context entry and exit. Unlike JCop, ECaesarJ does not provide a layer-like representation and composition mechanism of

behavioral variations. Furthermore, objects must explicitly handle context change, whereas event-based context implicitly changes the composition.

EventJava [EJ09] models events as asynchronous methods and compound events by *correlation patterns*. Event-specific behavior is encapsulated in the method bodies of correlation patterns that allow access to application-specific data and to implicit context information of the event. This can then be customized for application-specific purposes. The execution of event methods can be restricted through predicates specified in a `when` clause. Contrary, JCop's `when` pointcut specifies the constraints under which an event is triggered.

In *Ptolemy* [RL08], code blocks are bound to events, similar to pointcut-advice binding in AOP. Classes can contain binding definitions to such events or to compositions of multiple events. Events are explicitly announced, in contrast to JCop's implicitly evaluated `when`. Ptolemy's event handling mechanism provides the option for the immediate execution of functionality on event announcement. Contrary, event-based layer compositions in JCop do not immediate execute functionality but rather activate their corresponding layers on their next execution. That indirection assures that layer compositions wait until the execution stack has reached a safe point for recomposition, namely the execution of a layered method.

6.1.4 Reflective Layer Composition

Some situations require access to the currently active layers at run-time, as well as common layer definition and activation. A few COP languages, such as *ContextL* [CH05] and *ContextS* [HCH08] offer a reflective API that specifically allows the introspection of the current layer composition. Other languages, such as the Python-based implementations *PyContext* [vLDN07], *ContextPy* [HPSA10], and ContextJS offer access to the layer composition though the reflective API of their host language. However this functionality is not provided by an API but requires the use of internal functions of the COP implementations[17].

Moreover, ContextL's reflective API supports *reflective layer activation* that offers means to manipulate the run-time layer composition. JCop's reflective API presented in the appendix provides explicit access to layer composition, allows to modify compositions via composition handlers, and offers some auxiliary methods for the convenient expression of include and exclude relationships.

6.1.5 Dynamic Deployment of Context Classes

The *CaesarJ* [AGMO06] language is based on an alternative module concept by unifying classes, aspects, and packages. CaesarJ aspects can be deployed at run-time using different kinds of dynamic scope, much like context classes. CaesarJ also supports virtual classes [MMPN93]. This enables dynamic class extension, depending on the scope of the calling object. The ability of virtual classes to extend modules is similar to that of layers. However, class extension with layers is not bound on the caller's module but differs depending to the current layer composition.

6.2 Layer Declaration

In the following, we compare the properties of layer declarations of JCop to those of related languages.

[17] In Figure 6, we denote this reflective access with *(x)*.

6.2.1 Adaptation of Framework Code

The Objective-C based language *Subjective-C* [GCM+10] allows for the definition of partial methods whose base method source code is not accessible. Therefore, it can be used to adapt framework code with partial methods. Subjective-C does not provide control-flow based layer composition. Instead, layers are globally activated and deactivated. Because of this global activation, the problem of crosscutting layer composition, i.e., different control flows that need to be adapted with the same composition statement, does not occur in Subjective-C. We believe that global layer activation can only be applied to scenarios where the interruption of the current execution by an adaptation does not lead to inconsistent state.

The implementation of the layer composition makes use of the ability of Objective-C to change the virtual lookup tables at run-time. This implementation strategy for COP has been first proposed by the *cj* language [SHHJ09] and also implemented for JCop (using the dynamic invocation feature of Java 7) [AHH10].

6.2.2 Layer Guards and Implicit Layer Activation

The activation of a partial method might not only depend on one layer being active. Instead, there might be scenarios in which the activation of a partial method is only possible when a certain combination of layers is active. To declare such layer relationships, ContextPy provides the concept of *guards*. Guards are functions that receive the list of currently active layers and return a Boolean value indicating whether the partial method this guard was assigned to is to be activated.

Similarly, PyContext supports a kind of *implicit layer activation* that is designed to deal with the issue of scattered layer activations. Implicit layer activation factors out layer composition from the main program logic and, instead, defines a method returning whether the layer is active or not. Each time a layered method is called and the layer is registered for implicit activation, the `active` method is executed and its corresponding partial method, if necessary, contributes to the final composition.

Both approaches are similar to layer composition handlers in JCop. However, composition handlers can control and modify the entire layer composition list, whereas guards and implicit activation can only decide about their own participation.

6.2.3 Layer Relationships

ContextL provides a *feature description language library* [CD08] that supports the declaration of relationships between layers. Layer composition consistency with regard to the feature description is checked by an algorithm that computes valid composition combinations. If, at run-time, a layer composition violates the feature description, the conflict is solved interactively by the end-user. Therefore, a window dialog asks the end-user which feature should be selected (i.e, which layer should be activated). This approach is useful as long as a system's layers and the semantics of their composition are comprehensible for end-users. If layers are declared at a more technical level, end-users may not be able to decide about the composition. Furthermore, an interactive selection—and its interruption of the program execution—is not acceptable for some application domains.

The reflective API of JCop allows the declaration of 'include' and 'exclude' relations. These methods are implemented in two ways. First, a violation of a feature description throws a run time exception. In this case, the exception may be used to handle the inconsistency. Second, the `include` method can be parameterized with layers that should be activated. Similarly, `exclude` is allowed to deactivate the layer that

should not be part of the composition. With these methods, it is possible to define an API with similar behavior as the feature description implementation in ContextL. However, the ContextL implementation is more declarative than the imperative approach.

Layer relationships can also be expressed in EventCJ by its *layer transition rules* [KAM11]. For example, the following EventCJ code expresses that, if a layer `Rain` is active, it should be deactivated and replaced by a layer `Snow`:

```
direction SwitchWeather {
  declare event Snowfall() : after call(* *.snow());
  transition Snowfall : Rain switchTo Snow;
}
```

Note that the event `Snowfall` triggers the transition whenever the method (here: `snow`) is called. Contrary, JCop does not use this generic events approach but only accepts layer composition events. Layer transition rules also consider dynamic change to these rules. For example, to change this rule if the hero is solving the final quest (see Subsection 4.2), one can declare a second rule and event that implements the alternative behavior.

6.2.4 Abstract Partial Method Declarations

PyContext defines layers by a class inheriting from a class `Layer`. Therefore, partial methods that are meant to be abstract could throw a `TypeError` exception telling that they should be implemented by a subtype. However, this error would occur at runtime. In JCop as in Java, abstract methods and classes are checked at compile time, which is more convenient for the developer.

6.2.5 Static Active Layers

The maintenance and development of software product-lines is addressed by *feature-oriented programming* [BSR03] (FOP). Its Java-based implementation AHEAD [Bat04] tool suite supports the *Jakarta* language that extends Java with constructs such as class refinements for static feature-oriented composition. Layers in Jakarta are distinct files describing static class refinements. The modularization concepts of FOP and COP are similar: Both introduce new or alternative program behavior through features (FOP) or layers (COP), respectively. However, FOP applies compile-time composition of feature variations, while COP applies run-time composition of layers. Static active layers in JCop have the same semantics than to layers in FOP as they extend a system at compile time.

6.3 Related Language Approaches

Some other programming language approaches have been proposed which are closely related to COP. In the following, we introduce the most important ones.

6.3.1 Aspect-oriented Programming

AspectJ The AOP language *AspectJ* [KHH+01] established the notion of join points (events during the execution of a program) that can be described by pointcuts and can be adapted by advice blocks. AspectJ's `execution` pointcut is similar to JCop's `on` pointcut, except that the method patterns of `on` are restricted to public methods to preserve encapsulation.

AspectJ's `if` pointcut contains an expression that is evaluated at a join point. It is of use only when concatenated with other pointcuts that provide the set of join points on which `if` is evaluated. JCop's `when` predicate is similar to AspectJ's `if` as it dynamically evaluates a condition. However, `when` uses an implicit set of join points (all executions of layered methods). AspectJ's aspects are woven at compile or load time and are globally scoped in contrast to JCop's instantiable and dynamically deployable context classes.

The usage of `this` and `args` is different in JCop and AspectJ. In AspectJ, they can bind new join points, which makes sense as the AspectJ pointcut model is more generic than the JCop pointcut model. However, for the specific purpose of declarative layer composition, the AspectJ implementation would be more complicated. For example, the following pointcut uses a reference of type `A` to restrict the join points collected by `on` to method executions within `myA`.

```
contextclass Ctx {
  A myA = A();
  on (int pckg.A.x()) && this(myA) : ...
}
```

A similar expression in AspectJ would require the declaration of a second variable as advice parameter:

```
aspect Ctx {
  A myA = A();
  int around(A thisA) :
    execution (int pckg.A.x()) && this(thisA) && if(thisA == myA) {...}
}
```

The main distinction between AOP and COP is that the former allows for a joint specification of *when* during program execution (composition) *what*kind of functionality (adaptation) should be used. COP separates composition (e.g., using explicit `with` statement) from adaptation (e.g., using layers and partial methods).

Aspect-based COP Implementations Two COP approaches implement the context-aware dispatch by an aspect that expects special join point hooks in the program. *ContextLogicAJ* [AH08] is implemented based on the *LogicAJ* [KRH04] aspect language, while *JavaCtx* [SGP11] uses AspectJ. In both approaches, layers are represented by subclasses of a special layer class. Partial method declarations are implemented as Java methods in their host class that follow a specific naming convention. In ContextLogicAJ, the first partial method parameter represents the corresponding layer. In JavaCtx, the partial method name contains the corresponding layer name as suffix. Layer composition uses in both cases two methods that express the begin and the end of the composition scope. The actual context-aware method dispatch is implemented by an aspect that intercepts calls to layered methods and delegates them to the partial methods using the naming conventions. Both approaches require to consider some idioms (declaring a layer class that only serves as an identifier, adhering to the partial method naming conventions). Moreover, they do not provide a scoped layer composition. Instead, layer activation and deactivation must be explicitly expressed by an activation and deactivation method. Hence, layer composition is not scoped to the dynamic extent, which may lead to state inconsistencies.

6.3.2 Context-aware Programming

Us [SU96] supports *subjective programming* where message lookup depends not only on the receiver of a message, but also on a second object, called the perspective. The perspective allows for layer activation similar to ContextL. Us does not support implicit activation of layers.

The *Ambience* language is another approach to context-orientation, based on the *Ambient Object System* (AmOS) [GMC08]. AmOS is a prototype-based object system built on top of Common Lisp that supports behavioral adaptations with partial method definitions and context objects, which correspond to COP layers. It supports the implementation of behavioral variations by two language constructs, partial method definitions and context objects (which correspond to COP layers). Ambience does not support implicit context activation based on the evaluation of an expression as supported by JCop's `when` predicate.

The *Flute* language [BVDR+12] addresses the specification of interruptible context-de-pendent executions. In this model, any method execution can be interrupted or resumed at any type, e.g. by a context switch. Besides variations of procedures (*procedure modes*) for each context, Flute allows variables to contain different state in different contexts (*variable modes*). Modes can be grouped by *modals*. Context change is triggered by value changes of context sources, that are represented as *reactive values*. A reactive value employs a push-driven model, which means that any computation using that value is recomputed. This recomputation then uses the appropriate mode corresponding to the new value of the context source.

Context-aware aspects [TGDB06] can adapt the advice execution depending on the program's run-time state. This approach comes with a Java-based context model; context can be described by a pointcut-like API.

CSLogicAJ [RSC06] features context-sensitive service aspects to adapt service behavior. The extensible join point model supports context change. Special pointcuts can be used to describe context. An asynchronous advice type executes service adaptation triggered by these context-aware pointcuts. The service interception and adaptation is done by *Ditrios*, an OSGi-based middleware. In the current version, CSLogicAJ is restricted to local OSGi service bundles. Server-side services can not be intercepted.

7 Conclusions

Context-oriented programming (COP) addresses the representation of context-dependent behavioral variations, which often crosscut other concerns of an application. COP proposes layers as a way to support the adaptation of crosscutting behavioral variations. Contemporary implementations of COP provide the explicit composition of layers per control flow [CH05].

In previous experiments with COP languages and case studies, we identified application scenarios where not only the modularization of adaptations but also the adaptations' run-time composition are crosscutting concerns. As a consequence, we propose to separate both the *adaptation* and the *composition* code from the base.

Currently, in context-oriented languages, the modularization of adaptation code is achieved by layers. However, modularization of composition code has not been addressed. Instead, explicit composition statements (`with`, `without`) are defined in the base code and can tangle the core concerns of the base code. Moreover, a layer composition may be required at different source code location leading to a scattered

implementations. Therefore, we argue that layer composition may be a homogeneous crosscutting concern that deserves dedicated language support. Our JCop language design addresses crosscutting layer composition by aspect-oriented mechanisms called context classes.

In order to support modularization not only *by* layers but also *within* their declaration, layers offer inheritance mechanisms that are similar to that of Java classes. To access layer types, we created new keywords to access a layer instance (`thislayer`), a super layer instance (`superlayer`), and the partial method of a super layer (`superproceed`).

In addition, JCop is equipped with a feature-rich reflection API that enables reasoning about layers and their composition. It also supports the expression of layer dependencies, whereby layers can control their own activation and may add or remove other layers to or from a composition.

We evaluated JCop's new language constructs in application domains where previous COP approaches suffered from redundant composition statements. In particular, we applied JCop to framework-based applications such as RetroAdventure where source code modules affected by adaptations are relieved from composition code. Hence, the coupling between the base code and its adaptations is reduced. Therefore, behavioral variations can be represented in a concise and modular way.

References

[ADB+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a Better Understanding of Context and Context-Awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, Berlin, Heidelberg, Germany, 1999. Springer-Verlag. doi:10.1007/3-540-48157-5_29.

[AGMO06] 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. doi:10.1007/11687061_5.

[AH08] Malte Appeltauer and Robert Hirschfeld. Explicit Language and Infrastructure Support for Context-aware Services. In *Beiträge der 38. Jahrestagung der Gesellschaft für Informatik*, volume INFORMATIK 2008 - Beherrschbare Systeme dank Informatik of *Lecture Notes in Informatics*, pages 164–170, München, Germany, 2008. Gesellschaft für Informatik.

[AH12] Malte Appeltauer and Robert Hirschfeld. Declarative Layer Composition in Framework-based Environments. In *Proceedings of the 4th International Workshop on Context-Oriented Programming*, COP'12, New York, NY, USA, 2012. ACM Press. doi:10.1145/2307436.2307437.

[AHH10] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. Layered Method Dispatch with INVOKEDYNAMIC - An Implementation Study. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP'10, New York, NY, USA, 2010. ACM Press. doi:10.1145/1930021.1930025.

[AHHM11] 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):272–292, 2011.

[AHM09] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the Development of Context-dependent Java Applications with ContextJ. In *Proceedings of the 1st International Workshop on Context-Oriented Programming*, COP'09, New York, NY, USA, 2009. ACM Press. doi:10.1145/1562112.1562117.

[AHM+10] 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.

[ALS08] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008. doi:10.1109/TSE.2007.70770.

[Bat04] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society Press. doi:10.1109/ICSE.2004.1317496.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2003. doi:10.1109/TSE.2004.23.

[BVDR+12] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, pages 67–84, New York, NY, USA, 2012. ACM. doi:10.1145/2384592.2384600.

[CD08] Pascal Costanza and Theo D'Hondt. Feature Descriptions for Context-oriented Programming. In *Proceedings of the 12th International Conference on Software Product Lines, SPLC'08*, pages 9–14. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[CH05] 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. doi:10.1145/1146841.1146842.

[Dey01] Anind K. Dey. Understanding and Using Context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001. doi:10.1007/s007790170019.

[EJ09] Patrick Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP'09*, Lecture Notes in

Computer Science, pages 570–594, Berlin, Heidelberg, Germany, 2009. Springer-Verlag. doi:10.1007/978-3-642-03013-0_26.

[Gal06] Ben Galbraith. Developing Swing Applications. Sun Microsystems Technical Articles, 2006.

[GCM+10] Sebastián González, Nicolas Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing Context to Mobile Platform Programming. In *Proceedings of the 3rd International Conference on Software Language Engineering, SLE'10*, Lecture Notes in Computer Science, pages 246–265, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-19440-5_15.

[GMC08] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008. doi:10.3217/jucs-014-20-3307.

[GSS+06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006. doi:10.1109/MS.2006.24.

[HCH08] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-Oriented Programming with ContextS. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE 2007, Revised Papers*, Lecture Notes in Computer Science, pages 396–407, Berlin, Heidelberg, Germany, 2008. Springer-Verlag. doi:10.1007/978-3-540-88643-3_9.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008. doi:10.5381/jot.2008.7.3.a4.

[HHM07] Stefan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/Java Language Definition - Version 1.0. Technical Report 3, TU Berlin - Fakultät IV, 2007.

[HPSA10] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. In *Proceedings of the 25th Symposium on Applied Computing, SAC'10*, New York, NY, USA, 2010. ACM DL. doi:10.1145/1774088.1774546.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-oriented Programming*, 1(2):22–35, 1988.

[KAM10] Tetuso Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing Event-based Context Transition in Context-oriented Programming. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming, COP'10*, New York, NY, USA, 2010. ACM Press. doi:10.1145/1930021.1930023.

[KAM11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11,

pages 253–264, New York, NY, USA, 2011. ACM. doi:10.1145/1960275.1960305.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'01*, Lecture Notes in Computer Science, pages 327–354, Berlin, Heidelberg, Germany, 2001. Spinger-Verlag. doi:10.1007/3-540-45337-7_18.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP'97*, Lecture Notes in Computer Science, pages 220–242, Berlin, Heidelberg, Germany, 1997. Springer-Verlag. doi:10.1007/BFb0053381.

[KRH04] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable Pattern Implementations need Generic Aspects. Research report C-196, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan, 2004.

[LASH11] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming*, 76:1194–1209, 2011. doi:10.1016/j.scico.2010.11.013.

[MMPN93] Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[NC09] Nokia Corporation. Qt 4.6 Whitepaper, 2009. http://qt.nokia.com/files/pdf/qt-4.6-whitepaper *(visited: 2013-06-01)*.

[NNG09] Angel Núñez, Jacques Noyé, and Vaidas Gasiũnas. Declarative Definition of Contexts with Polymorphic Events. In *Proceedings of the 1st International Workshop on Context-Oriented Programming*, COP'09, New York, NY, USA, 2009. ACM Press. doi:10.1145/1562112.1562118.

[Ope] Open Handset Alliance. Android Developers Platform. http://developer.android.com *(visited: 2013-06-01)*.

[RAL+11] Tobias Rho, Malte Appeltauer, Stephan Lerche, Armin B. Cremers, and Robert Hirschfeld. A Context Management Infrastructure with Language Integration Support. In *Proceedings of the 3rd International Workshop on Context-oriented Programming.*, COP'11, New York, NY, USA, 2011. ACM Press. doi:10.1145/2068736.2068739.

[Red11] RedHat inc. JBoss, 2011. http://www.jboss.com *(visited: 2013-06-01)*.

[RL08] Hridesh Rajan and Gary T. Leavens. Ptolemy: A Language with Quantified, Typed Events. In *Proceedings of the 22nd European conference on Object-Oriented Programming, ECOOP '08*, pages 155–179, Berlin, Heidelberg, Germany, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_8.

[RSC06] Tobias Rho, Mark Schmatz, and Armin B. Cremers. Towards Context-Sensitive Service Aspects. In *Proceedings of the Workshop on Object Technology for Ambient Intelligence and Pervasive Computing, OT4AmI'06*, Berlin, Heidelberg, Germany, 2006. Springer-Verlag.

[SAH11] Christopher Schuster, Malte Appeltauer, and Robert Hirschfeld. Context-oriented Programming for Mobile Devices: JCop on Android. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming, COP'11*, New York, NY, USA, 2011. ACM Press. doi:10.1145/2068736.2068741.

[SGP11] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. JavaCtx: Seamless Toolchain Integration for Context-oriented Programming. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming, COP'11*, New York, NY, USA, 2011. ACM. doi:10.1145/2068736.2068740.

[SHHJ09] Hans Schippers, Michael Haupt, Robert Hirschfeld, and Dirk Janssens. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proceedings of the 24th Symposium on Applied Computing, SAC'09*, New York, NY, USA, 2009. ACM Press. doi:10.1145/1529282.1529716.

[SU96] Randall B. Smith and David Ungar. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996. doi:10.1002/(SICI)1096-9942(1996)2:3<161::AID-TAPO3>3.0.CO;2-Z.

[TGDB06] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware Aspects. In *Proceedings of the 5th International Symposium on Software Composition, SC'06*, Lecture Notes in Computer Science, Berlin, Heidelberg, Germany, March 2006. Springer-Verlag. doi:10.1007/11821946_15.

[vLDN07] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented Programming: Beyond Layers. In *Proceedings of the International Conference on Dynamic Languages, ICDL'07*, pages 143–156, New York, NY, USA, 2007. ACM Press. doi:10.1145/1352678.1352688.

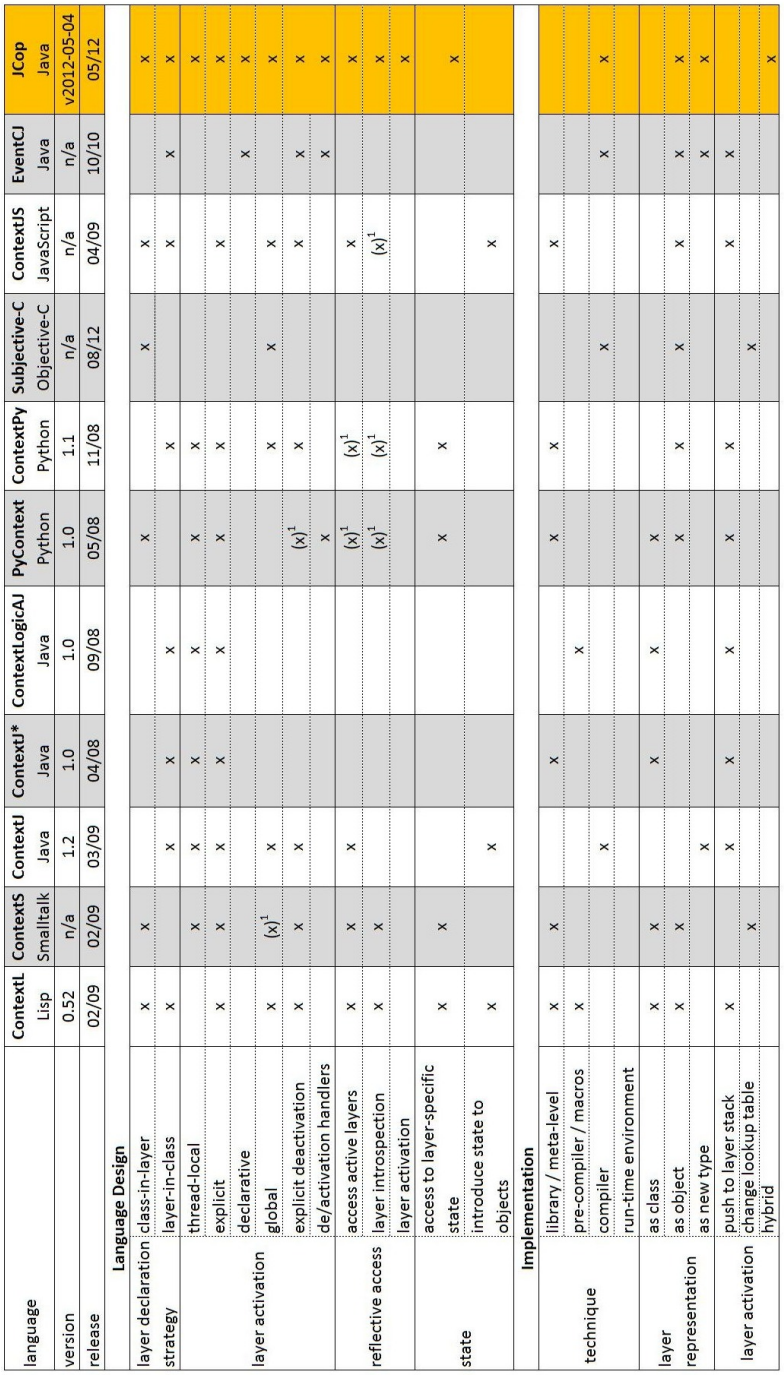
About the authors

Malte Appeltauer is a researcher and developer at the SAP Innovation Center. He received a Ph.D. in Computer Science at the Hasso-Plattner-Institute, University of Potsdam as a member of the HPI Research School on Service-Oriented Systems Engineering. He can be reached at malte.appeltauer@sap.com. See also www.hpi.uni-potsdam.de/swa/publications.

Robert Hirschfeld is a Professor of Computer Science at the Hasso-Plattner-Institut at the University of Potsdam. He received a Ph.D. in Computer Science form the Technical University of Ilmenau, Germany. He can be reached at hirschfeld@hpi.uni-potsdam.de. See also www.hpi.uni-potsdam.de/swa/people/hirschfeld.

Jens Lincke is a research assistant at the Hasso-Plattner-Institute, University of Potsdam. He can be reached at jens.lincke@hpi.uni-potsdam.de. See also www.hpi.uni-potsdam.de/swa/people/lincke.

A Comparison of COP Languages



language		ContextL Lisp	ContextS Smalltalk	ContextJ Java	ContextJ* Java	ContextLogicAJ Java	PyContext Python	ContextPy Python	Subjective-C Objective-C	ContextJS JavaScript	EventCJ Java	JCop Java
version		0.52	n/a	1.2	1.0	1.0	1.0	1.1	n/a	n/a	n/a	v2012-05-04
release		02/09	02/09	03/09	04/08	09/08	05/08	11/08	08/12	04/09	10/10	05/12
Language Design												
layer declaration	class-in-layer	x	x				x		x	x		x
strategy	layer-in-class	x		x	x	x		x		x	x	x
layer activation	thread-local		x	x	x	x	x	x				x
	explicit	x	x	x	x	x	x	x		x		x
	declarative										x	x
	global	x	(x)[1]	x				x	x	x		x
	explicit deactivation	x	x	x			(x)[1]	x		x	x	x
	de/activation handlers						x				x	x
reflective access	access active layers	x	x	x			(x)[1]	(x)[1]		x		x
	layer introspection	x	x				(x)[1]	(x)[1]		(x)[1]		x
	layer activation											x
state	access to layer-specific state	x	x				x	x				x
	introduce state to objects	x		x						x		
Implementation												
technique	library / meta-level	x	x		x		x	x		x		
	pre-compiler / macros	x				x						
	compiler			x					x		x	x
	run-time environment											
layer representation	as class	x	x		x	x	x					
	as object	x	x				x	x	x	x	x	x
	as new type			x							x	x
layer activation	push to layer stack	x		x	x	x	x	x		x	x	
	change lookup table		x						x			
	hybrid											x

[1] Functionality only accessible via the host languages meta-level features

Figure 6 – Feature comparison of several COP languages.

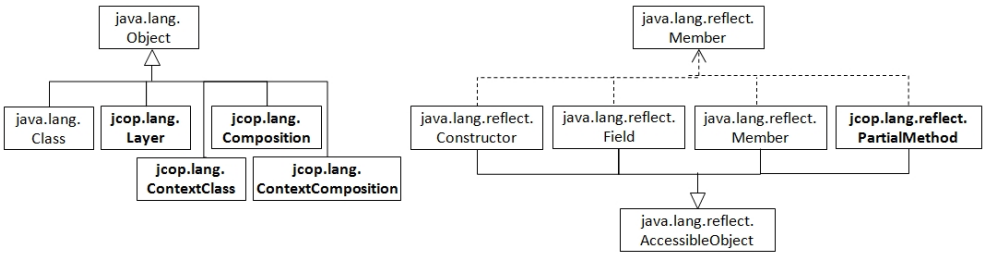


Figure 7 – Extension of the Java reflection API.

B JCop 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 7.

B.1 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 include(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 exclude(Layer toBeExcluded, boolean stopOnConflict)`

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 void excludeAll(Class<Layer> toBeExcluded, boolean stopOnConflict)`

Specifies that on layer activation no instance of the Layer `toBeExcluded` is part of 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.

B.2 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.

B.3 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.

B.4 jcop.lang.ContextComposition

`public static ContextComposition current()`

Returns the thread-local context composition.

`public ContextClass[] getContextClasses()`

Returns the deployed context class instances.

B.5 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

B.6 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.