Event-Specific Software Composition in Context-Oriented Programming

Malte Appeltauer[1], Robert Hirschfeld[1], Hidehiko Masuhara[2], Michael Haupt[1], and Kazunori Kawauchi[2]

[1] Hasso-Plattner-Institute, University of Potsdam, Germany
first.last@hpi.uni-potsdam.de
[2] Graduate School of Arts and Science, University of Tokyo, Japan
{masuhara,kazu}@graco.c.u-tokyo.ac.jp

Abstract. Context-oriented programming (COP) introduces dedicated abstractions for the modularization and dynamic composition of crosscutting context-specific functionality. While existing COP languages offer constructs for control-flow specific composition, they do not yet consider the explicit representation of event-specific context-dependent behavior, for which we observe two distinguishing properties: First, context can affect several control flows. Second, events can establish new contexts asynchronously. In this paper, we propose new language constructs for event-specific composition and explicit context representation and introduce their implementation in JCop, our COP extension to Java.

1 Introduction

With the increasing demand for personalization and mobility of applications, context awareness gains growing relevance as a distinguishing feature of software systems. To meet the challenges of developing and managing context-specific behavior, several approaches have emerged, each providing its own definition of context. We adopt a notion where context is constituted by a set of predicates and a set of variation modules. The former are evaluated to determine the context's presence, and the latter are composed based on the result of predicate evaluation. Variation implementations are often scattered over application source code and can so be characterized as crosscutting concerns. With that, a major task of context representation is the modularization of such crosscutting concerns. In addition to modularization, context-specific crosscutting concerns require means for dynamic composition.

Context-oriented programming [22] (COP) is an approach to representing context-specific concerns, focusing on dynamic composition of control flows. COP allows for the definition of *layers*, modules that crosscut object-oriented decomposition and encapsulate the implementation of behavioral variations. For instance, a security layer can extend various methods with access control features without affecting the original method declarations. Depending on the execution context, layers are *composed* into a system at run-time. A layer composition defines the order in which layers adapt the base system. This way, COP separates

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 50–65, 2010.

the *definition* of adaptations from their *composition*, distinguishing it from alternative multi-dimensional modularization techniques such as aspect-oriented programming [26] (AOP), Mixins [12], or Classboxes [11]. The aforementioned security layer could be applied for specific control flows, while at the same time other computations can be executed with the basic functionality.

In the following, we distinguish between the separation of *adaptation code* and *composition code* from the base code. While the former is adequately handled by layers, the latter deserves better language support. The COP languages implemented so far [2] support selective activation and deactivation of layer compositions, expressing programmatically when the application enters and leaves certain contexts. It is, however, not enough to regard context as being entirely under programmer control; instead, context can impose itself on the running application "from the outside". Based on this observation, we distinguish control-flow specific from event-specific contexts. Two key properties characterize them:

1. Event-based context can overlap several control flows, unlike control-flow specific context, which is confined to a single control flow. For instance, context change in graphical user interface (GUI) applications can affect the behavior of several event handler methods at once.
2. Event-based context entry and exit often cannot be localized at fixed points in the control flow. Instead, context entry depends on asynchronous events independent from main control flow. Moreover, a certain context is often active until another event changes the composition. Any kind of sensor data, such as localization or temperature, are examples of independent context information that may asynchronously trigger system recomposition.

The former property implies that event-based context (de)activation leads to layer composition statements' being scattered over several locations, each of which corresponds to one of the affected control flows. The latter property implies that it is impossible to determine the locations where to place layer composition (de)activation. Also, asynchronous composition can lead to inconsistent system state within a control flow. With the abstractions of state-of-the-art COP languages, event-based context (de)activation cannot be represented without scattering layer composition statements over the program. Instead, first-class support for contexts is required, enabling declarative description of events that constitute context entry and exit. In addition, a possible solution must take composition consistency of asynchronous context change into account.

Contribution. In this paper, we motivate the need for explicit representation of event-specific context-dependent composition along a case study that we conducted using ContextJ [5, 3], our earlier COP extension to the Java programming language. We present appropriate abstractions adopted from AOP to cope with event-based behavioral variations. We introduce the *JCop* programming language extension that supports these constructs while preserving composition consistency as defined by COP. As a proof of concept, we apply JCop to our case study and discuss its expressiveness.

Outline. The rest of the paper is structured as follows. Section 2 introduces COP and describes our case study in which we developed a context-aware event-based GUI application using ContextJ. We discuss our experience concerning the case study in Section 3. Section 4 introduces JCop, Section 5 discusses the expressiveness of JCop with respect to the GUI implementation. Section 6 presents related work, while Section 7 summarizes the paper.

2 Event-Specific Behavioral Variations

Any computation in a program flow is executed within a specific context, such as system state or user-specific configuration, that can influence system behavior. The COP approach provides a first-class representation of context-specific behavioral variations that can be dynamically composed for a specific control flow. COP focuses on control-flow specific composition of behavioral variations and omits providing dedicated abstractions for event-specific composition, which we will address in the next sections.

2.1 Context-Oriented Programming

COP extends object-oriented programming with first-class abstractions for behavioral variations that can be composed into a system depending on execution context. COP assumes *context* to be *everything that is computationally accessible*, such as object state, network bandwidth, or user interaction.

COP provides *layers* [9] as a modularization concept that can crosscut an object-oriented decomposition and encapsulate context-specific behavioral variations, represented as *partial method definitions*. COP extends object-oriented method dispatch with dynamic composition of crosscutting concerns. To distinguish between the different kinds of method definitions, we introduce the terms *plain method definition* and *layered method definition*. A plain method is one whose execution is not affected by layers. Layered methods consist of a *base method definition*, which is executed when no active layer provides a corresponding partial method, and at least one partial method definition.

Layers can be activated and *composed* with others at run-time. When activated, layered method calls are dispatched to the partial method provided by the layer. Partial methods can be executed before, after, or around the base method definition. In a composition, multiple layers may provide partial definitions of the same method. In that case, a partial method can *proceed* to the next partial definition in the composition or, if none exists, to the base method definition. This feature has been previously introduced by other languages such as Common Lisp [24] and AspectJ [25]. Layer composition is controlled *per thread* and is by default scoped to the dynamic extent of a block of statements.

ContextJ[1] [2, 3] is a COP implementation for Java. It supports layer declaration within classes and explicit layer composition. In the following, we will explain ContextJ's main features along an example.

[1] ContextJ is available for download at `http://www.hpi.uni-potsdam.de/swa/cop`

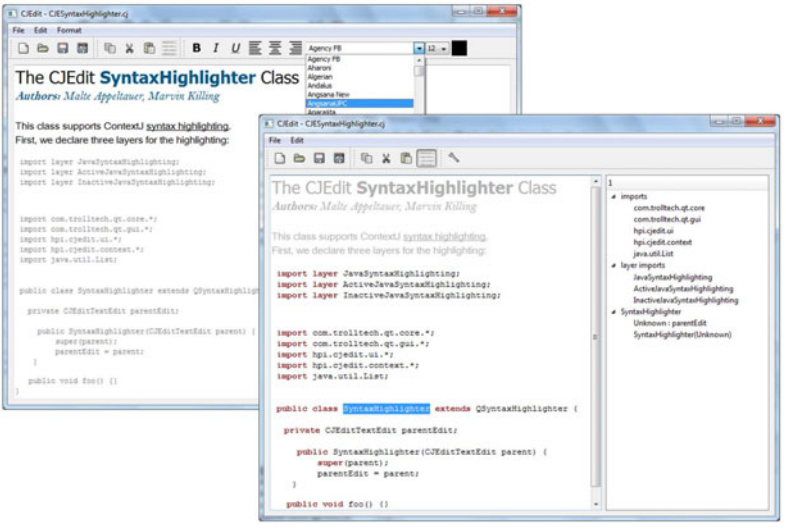

Fig. 1. Screenshots of CJEdit. *Left*: rich-text editing with format toolbars and menus. *Right*: program development is supported by an outline and focus on source code blocks.

2.2 Case Study: CJEdit

As a case study [4], we have developed a little IDE using ContextJ, whose GUI provides context-specific behavior.

Figure 1 shows two screenshots of our *CJEdit* application, a simple programming environment that provides different UI elements and behavior for the user-driven activities *programming* and *commenting*. The left-hand screenshot presents the application's commenting mode in which the toolbar offers various text formatting actions. The right-hand image shows the programming mode, where the editor comes with an outline and a different toolbar. To support focusing on source code, any rich text within the document is displayed in gray. The editor supports syntax highlighting, an outline view, a compilation/execution toolbar, and *rich text commenting features*, such as font and color modifications. Based on the user's actual task (i. e., context), the UI only offers relevant tools, menus, and widgets. The UI is recomposed upon context switches, which are either directly triggered by the user, or by text cursor changes. To enter programming context, the user can push a toolbar button. Moreover, context is changed whenever the text cursor moves from text to code and vice versa. CJEdit's core is implemented using ContextJ and the *Qt Jambi GUI Framework*[2]. The editor consists of approximately 3,000 lines of code in 38 classes.

Figure 2 shows the implementation of the *programming* activity-specific widgets using layers. In ContextJ, layers, denoted by the keyword `layer`, can be

[2] Nokia Corporation, Whitepaper: A Technical Introduction to Qt, 2008 `www.qtsoftware.com`

```
import layer CodeWidgets;
import layer Outline;
import layer RTFWidgets;

public class CJEditWindow extends QMainWindow {
  ...
  private void drawToolBars() {...}
  private void drawMenus() {...}
  private void drawWidgets() {... drawMenus(); drawToolbars(); }

  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() {...}
  }
  layer Outline {
     ...
  }
  layer RTFWidgets {
      ...
  }
}
```

Fig. 2. Layered specification of task-dependent GUI Widgets

defined in classes[3] and contain partial method definitions that are executed—depending on their modifiers—before, after, or around their base method.

The same layer can be partially defined in multiple classes; for instance, `CodeWidgets` can also provide partial methods for `CJEditTextEdit`, which implements the text editor widget. The layers shown in Figure 2 provide partial methods responsible for drawing UI elements, and auxiliary methods accessible from within the layer only to create these objects.

Each text block object of the underlying document tree holds a list of layers that should be activated when its text is focused by the user. A focus is set by text cursor selection. By default, text blocks refer to the layers responsible for *rich text commenting* behavior. If the user switches to the *programming* activity (by pressing the code button in the toolbar), subsequently created text blocks are linked with programming environment-specific layers.

The application is recomposed and its GUI redrawn whenever the current block type switches. The dynamic composition of our previously specified layers is depicted in Figure 3. For layer composition, ContextJ provides a `with` statement that specifies the layers to be activated, and the dynamic extent for which the composition is valid. To explicitly disable a layer for a control flow, the `without` statement can be used. Recomposition can be triggered by the `onCursorPositionChanged` event handler that checks if the block type of the

[3] Note that this is a design decision in ContextJ and not a general restriction of COP. The JCop language introduced in Section 4 allows for the specification of layers as top-level elements.

```
public class CJEditWindow extends QMainWindow {
  private List<Layer> getLayersOfCurrentBlock() {
    if (currentBlock.getType() == BlockType.TEXT)
      // returns RTFWidget
    if (currentBlock.getType() == BlockType.CODE)
      // returns CodeWidget and Outline
  }
  private boolean blockTypeChanged() {
    // true if the focused block has a different type than its predecessor
    ...
  }
  void onCursorPositionChanged() {
    if (blockTypeChanged()) {
      with (getLayersOfCurrentBlock()) { drawWidgets(); }
    }
  }
  void onPrint() {
    with (getLayersOfCurrentBlock()) { ... }
  }
  void onSave() {
    with (getLayersOfCurrentBlock()) { ... }
  }
  void onFileNew() {
    with (getLayersOfCurrentBlock()) { ... }
  }
}
```

Fig. 3. Dynamic composition in CJEdit

previously focused block is different to that of the current block. If so, the method calls `drawWidgets` to update the UI using the current block's layer composition.

3 Lessons Learned

Although CJEdit is a relatively small application with only a few context-dependent concerns, its ContextJ-based implementation eases the development process compared to a plain Java solution. From a structural point of view, layers allow for a better separation of concerns. Base methods only have to care for the editor's default behavior, while layers completely encapsulate their context-specific variations. In our scenario, context-specific behavior is strongly coupled with private state of extended classes, thus layer declaration within classes is the appropriate strategy for layer implementation. Dynamic GUI adaption is also expressed naturally by layer compositions.

These benefits aside, some characteristics of GUI-based programming had to be considered that led to additional challenges for the ContextJ-based implementation. In the following, we discuss the two most important findings.

3.1 Problems

Scattered Composition Statements. User interaction with a GUI is event-driven rather than control flow-centric. This complicates dynamic extent-based layer composition as originally proposed by COP. Figure 4 depicts an execution sequence in CJEdit, where user interaction triggers several event handlers, such

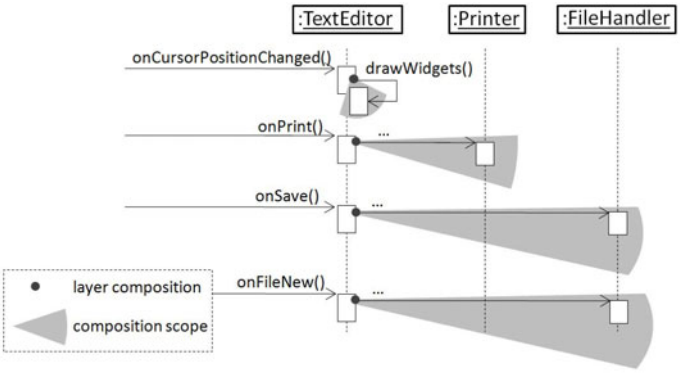

Fig. 4. Scattered layer compositions for event handlers in CJEdit

as printing the document, writing new text, or moving the text cursor through the document. Each event handler activates the layers of the currently focused text block for their respective control flows. In the source code, this issue is manifested in the form of identical `with` statements occurring in several event callback methods, as shown in Figure 3. More formally, we identify two different kinds of cross-cutting concerns, according to [1]: The actual behavioral variations implemented using layers are *heterogeneous* concerns, therefore they should be defined close to their respective objects. ContextJ serves this purpose well. Conversely, layer composition statements in CJEdit constitute a *homogeneous* cross-cutting concern that is not modularized by COP abstractions.

Event-specific Context Representation. COP generally defines context as *everything that is computationally accessible*, meaning that any event a system can recognize can influence the current layer composition. With the intention to avoid further restrictions, COP does not provide means for explicitly describing when context influences an execution. In general, it is impossible to globally describe the circumstances under which a composition should be used, since, for different executions, these properties can entirely change. The explicit `with` statement in COP exists due to this fact. However, the nature of what we denote as *event-specific context* is more predictable. In CJEdit, the *programming* context is constituted by the fact that the focused text contains source code; for the commenting context, the text must contain rich text elements. However, it is impossible to simply express these properties using the original COP abstractions provided by ContextJ. Instead, the application must provide information about the composition to be activated and must check for the context change itself.

3.2 Our Solution

To address the aforementioned problems, we propose an alternative to explicit composition statements. A declarative specification supporting the description

of control flow entry points helps avoid scattered composition statements. For event-based composition, we suggest a declarative definition of the event condition and its respective layer composition.

4 Event-Specific COP with JCop

From our CJEdit experiments, we conclude that scattered `with` statements and event-based layer composition deserve appropriate lingual abstractions. In this section, we present the JCop language that provides new constructs for declarative and event-based layer composition and a first-class event-based context representation. We discuss issues regarding module consistency within a dynamic extent and explain how JCop ensures consistent event-specific adaptations.

4.1 JCop Overview

The JCop language combines COP features provided by its predecessor ContextJ with alternative layer declaration and composition features.

Layers can either be defined within the classes for which they provide behavioral variations (*layer-in-class*), or in a dedicated top-level layer similar to an aspect (*class-in-layer*)[4] [22, 2]. Besides the structural differences of the two declaration styles, layer-in-class can access and extend the host object's internal state and methods, we restrict class-in-layer to public interfaces in order to sustain encapsulation. Developers can decide per situation if they prefer to define a layer within its enclosing class, allowing private member access, or to declare all partial definitions of a layer as one layer module to reduce scattering. For layer composition, JCop provides the control-flow specific `with` and `without` statements known from ContextJ.

The JCop compiler is implemented based on the *JastAdd* [20] meta-compiler framework and extends the Java 1.5 specification *JastAddJ* [16]. In addition, we adopted the AspectJ method pattern grammar used by the `abc` compiler [7]. It compiles JCop source code to Java 1.5 byte code.

4.2 Declarative Layer Composition

As stated above, event-based systems can handle multiple events whose behavior depends on identical layer composition. A layer composition spanning several control flows requires an explicit composition statement at the beginning of each of them. Thus, layer composition can be a homogeneous crosscutting concern applying the same functionality at several points in the system.

JCop introduces a *declarative layer composition* statement. It consists of a logic concatenation of *predicates* and a *composition block*. A composition

[4] If both styles are used to define the same layer, the compiler avoids ambiguities by asserting that a partial method must not be defined in both a class-in-layer and a layer-in-class declaration simultaneously.

```
1  in(CJEditWindow win) &&
2  (
3    on(* CJEditWindow.onPrint(..)) ||
4    on(* CJEditWindow.onSave(..)) ||
5    on(* CJEditWindow.onFileNew(..)) ||
6    on(* CJEditWindow.drawWidgets(..))
7  )
8  {
9    with(win.getLayersOfCurrentBlock());
10 }
```

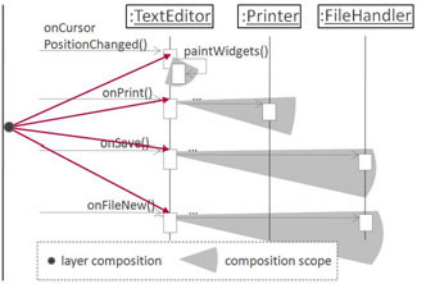

Fig. 5. Using the `on` predicate in CJEdit

block contains a `with` and/or `without` statement specifying the layers to be (de)activated[5]. Like for the general `with` statement, any expression returning a layer or a list of layers is a valid parameter, so layer compositions can also be computed. Declarative `with` statements are re-evaluated for every execution of the methods they are bound to.

Quantify Over Control Flows. To address the issue of scattered `with` statements, we adopted features of AOP [26, 18], where scattered functionality is expressed by *advice* blocks bound to *pointcuts* quantifying over a set of *join points*, well-defined events in the execution graph. JCop introduces a pointcut designator denoted `on`. It contains an AspectJ-like method pattern [25] specifying those methods to whose dynamic extent a layer composition is to be applied. We restrict the pattern to describe only those methods visible to the declaration without breaking encapsulation rules. The optional `in` predicate allows for binding the object on which the composition declaration should be evaluated.

Figure 5 presents a declarative layer composition that specifies the scope of layer activation for CJEdit. For all event handler callbacks and `createWidgets`, the composition of the currently focused block is used.

Conditional Composition. For a clearly specifiable set of method executions participating in a layer composition, the `on` predicate is our preferred means. For more complex structures, however, the explicit specification of control-flows becomes increasingly verbose.

In addition to `on`, JCop allows for a more implicit description of composition time independent of the actual execution in the main control flow. We support the fact that a context activation event is reflected in the change of some property that *is computationally accessible* and provide a `when` predicate that allows for the specification of a Boolean expression evaluating this property. The predicate is evaluated before the execution of any layered method that is potentially

[5] If a layer is referred by both lists, it is activated and deactivated at the same time and thus will be ignored for composition.

```
1  import layer RTFWidgets;
2  import layer CodeWidgets;
3  import layer Outline;
4
5  context Commenting {
6    in(CJEditWindow win) &&
7    when(win.getCurrentBlockType() ==
8         BlockType.Commenting)
9    {
10       with(RTFWidgets);
11       without(CodeWidgets, Outline);
12   }
13 }
14 context Programming {
15   in(CJEditWindow win) &&
16   when(win.getCurrentBlockType() ==
17        BlockType.Programming)
18   {
19       with(CodeWidgets, Outline);
20       without(RTFWidgets);
21   }
22 }
```

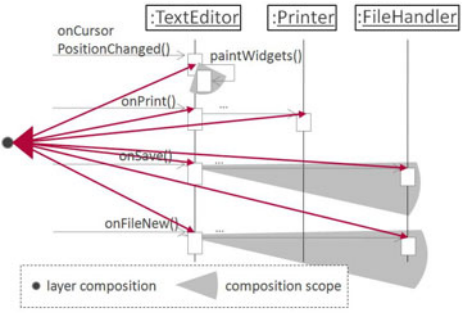

Fig. 6. Using the `when` predicate in CJEdit

affected by the respective layer composition. If the `when` predicate is evaluated to `true`, these layered methods are executed using the composition.

Figure 6 shows two declarations of event-based composition for CJEdit (Lines 6–12, 15–21). In both statements, the `when` predicates specify the current block type required to activate the composition. The predicate expressions are evaluated on a `CJEditWindow` instance that is bound by an `in` designator.

The `when` predicate completely relieves the developer from having to specify *where* a layer composition should begin and only requires the declaration of *when* composition takes place. Nevertheless, the combination of `on` and `when` is useful to restrict the scope in which `when` should be evaluated.

4.3 First-Class Context Representation

Since declarative and event-based composition statements are independent of specific objects, they should be defined in a dedicated location. For this reason, JCop provides a first-class `context` construct. Like layers, contexts are special singleton types that cannot be instantiated. The construct can host both declarative and event-based composition statements and auxiliary methods and fields.

Figure 6 presents context declarations for our programming and commenting contexts. The contexts contain an event-based composition statement that declares when the context changes and which layers are composed.

4.4 Composition Consistency in a Dynamic Extent

Programming languages and frameworks that support dynamic recompositions offer extended expressiveness. This, in turn, can lead to inconsistent and unintuitive control flows. A typical example is the recomposition of a method while

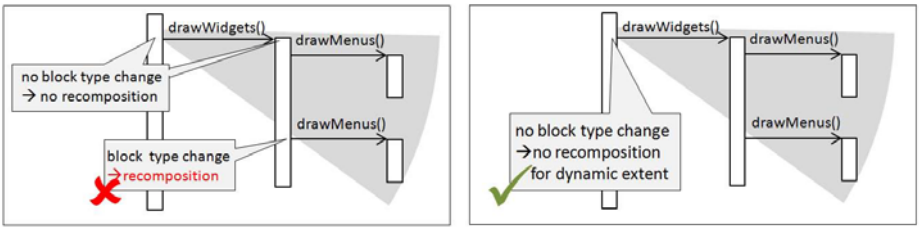

Fig. 7. Layer composition scopes and activation

it is being executed. To aid the developer in avoiding such undesired behavior, the original COP approach restricts layer composition to a dynamic extent. Our event-based composition evaluates the `when` predicate every time a method invocation is potentially dispatched to a layer involved in the composition. Without additional restrictions, this approach cannot guarantee that a dynamic extent is executed with a consistent layer composition.

Figure 7(a) exemplifies this issue for CJEdit. Assume that text focus can be changed asynchronously to the UI `drawWidgets` operation, which, among others, calls `drawMenus`. Both methods are layered and draw the UI according to the current context. If we apply the contexts declared in Figure 6 using the strategy described above, and block focus changes during `drawWidgets`, the redrawn GUI would partially consist of both programming and commenting context parts.

Besides the fact that such behavior is obviously undesired, the implicit and asynchronous composition activation is hard to debug. Tracing this kind of failures is tedious. JCop prevents such inconsistencies by ensuring that, once `when` is evaluated, the predicate will not be re-evaluated within the dynamic extent originating from this evaluation, as depicted in Figure 7(b). 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. In our extension, this assumption holds: a composition is valid until the control flow returns to the point at which the composition has been created.

5 Discussion

In Section 3, we identified some issues concerning the representation of event-based behavioral variations using programming language abstractions provided by the original COP approach. First, if layer compositions range over several control-flows, their respective composition statements must be repeated at any of their potential entry points. Second, event-specific composition cannot be explicitly declared but must be handled by application logic. In the following, we discuss how our new JCop language constructs solve these issues.

Composition Declarations to Solve Scattering. The phenomenon of scattered functionality that requires code repetition is well known as crosscutting

concern for which AOP provides encapsulation mechanisms. In JCop, we fuse COP with some concepts of AOP to address this problem.

JCop introduces context types that contain a declarative composition statement similar to a pointcut-advice construct. Declarative compositions allow for the layer composition of several control flows. Scattered composition statements can be avoided using the declarative composition in combination with the `on` predicate, which allows for the specification of all method executions to be included in the scope of a layer composition.

When-Declarations for Event-based Composition. In addition to `on`, JCop provides a `when` predicate for composition declarations. It describes the property that must be fulfilled for the activation of a composition and thus reliefs the application logic from managing compositions and their respective events. One of the key properties of COP is the consistency of a layer composition within a dynamic extent. JCop ensures that this property is not violated by event-based composition, as described in Section 4.4.

Context Types Encapsulate Context Specification. JCop's context types release the application logic from handling layer compositions and event-specific context changes. Besides composition declarations, context types can contain auxiliary members to compute layer compositions or store relevant context information.

6 Related Work

Other COP Languages. Most COP extensions have been developed for dynamic languages, such as Lisp [13, 14] Smalltalk [21], Python [31, 23], and the *delMDSOC* kernel [29]. They all implement the original semantics of COP, based on meta-programming facilities of their respective host language. A detailed comparison of COP language features is provided in [2]. JCop is the first language that fuses COP with AOP for a more declarative composition scope specification. Except for the Python extension *PyContext* [31] that provides implicit layer activation, none of the mentioned languages support event-based layer activation. The *Ambience* language is another approach to context-orientation. Based on the *Ambient Object System* [19], it supports behavior adaptations with 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.

Aspect-oriented Programming. The main distinction between AOP and COP (including JCop) is that the former allows for a joint specification of *when* in the execution flow *what* kind of functionality should be used, while COP separates *when* (using explicit `with` statements) from *what* (using layers and partial methods). JCop exceeds COP by introducing declarative composition statements.

AspectJ [25] is a popular Java language extension that established the notion of *join points*, well-defined events in the execution of a program that can be described by *pointcut* predicates and can be adapted by *advice* blocks. JCop's `on` predicate is equivalent to AspectJ's `execution` pointcut, except that the former's

method patterns 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 `if` as it dynamically evaluates a condition. However, `when` uses an implicit set of join points, namely all executions of layered methods.

Most AspectJ-like languages do not support dynamic aspect weaving that could simulate COP layer activation. However, they can mimic COP behavior using pointcuts and advice, though in an unwieldy manner since the pointcut specifications get complex. In some languages, such as *CaesarJ* [6], aspects can be deployed for a dynamic extent at run-time, much like explicit `with` statements. However, CaesarJ does not provide first-class context and behavioral variations but rather supports variability at a different level of abstraction.

Alternative Adaptation Techniques. Modularization approaches such as *traits* [30,15] and *mixins* [12] allow for an additional inheritance relationship next to the class hierarchy, but do not offer dynamic adaptation like layers. Feature-oriented programming (FOP) [10] and its Java-based implementation AHEAD [8] provide layer-like modules to specify adaptations of methods and classes (and other software artifacts). However, FOP and AHEAD apply compile-time composition of feature variations in contrast to run-time composition as provided by COP and JCop. The *Classbox/J* [11] module system extends Java's packaging and scoping mechanism. A classbox is an explicitly named scope in which classes and their members can be defined. Besides common subclassing, Classbox/J supports local refinement of imported classes by adding or modifying their features without affecting the originating classbox, much like layers and partial methods. However, it does not provide means for dynamic composition.

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

The CaesarJ extension *ECaesarJ* [27] supports the definition of context as a class implementing two events representing context entry and exit. Unlike JCop, ECaesarJ does not provide a layer-like representation and composition mechanism of behavioral variations. Moreover, objects must explicitly handle context change, whereas event-based context implicitly changes the composition.

EventJava [17] models events as asynchronous methods and compound events by correlation patterns. Event-specific behavior is encapsulated in method bodies of correlation patterns that allow access to application-specific data and to implicit context information of the event, which can 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` construct specifies the constraints under which an event is triggered.

In *Ptolemy* [28], 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, contrary to JCop's implicitly evaluated `when`. Ptolemy's event handling mechanism allows for the immediate execution of functionality on event announcement, while JCop ensures that event-based layer compositions wait until the execution stack has reached a safe point for recomposition.

7 Summary and Conclusion

We discussed the requirements for context-oriented programming languages to support event-specific context-dependent behavioral variations along a case study implemented using a conventional COP language. For a better separation of layer composition from application logic, we adopted pointcuts from AOP and developed declarative composition statements. As an implementation of these concepts, we presented JCop, our new language extension to Java. We applied JCop to our case study to show that our new language abstractions allow for a more declarative and intuitive specification of event-based behavioral variations.

As we have shown in this paper, different types of context require different programming language representations in its support. In future work, we will conduct a thorough analysis of variations of possible layer composition scopes beyond control-flow and event-based scope.

References

1. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Transactions on Software Engineering 34(2), 162–180 (2008)
2. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A Comparison of Context-oriented Programming Languages. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–6. ACM Press, New York (2009)
3. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ - Context-oriented Programming for Java (2009) (submitted)
4. Appeltauer, M., Hirschfeld, R., Masuhara, H.: Improving the Development of Context-dependent Java Applications with ContextJ. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–5. ACM Press, New York (2009)
5. Appeltauer, M., Hirschfeld, R., Rho, T.: Dedicated Programming Support for Context-aware Ubiquitous Applications. In: UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, Washington, DC, USA, pp. 38–43. IEEE Computer Society Press, Los Alamitos (2008)
6. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of CaesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)

7. Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An Extensible AspectJ Compiler. In: AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp. 87–98. ACM Press, New York (2005)
8. Batory, D.: Feature-Oriented Programming and the AHEAD Tool Suite. In: ICSE 2004: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, pp. 702–703. IEEE Computer Society, Los Alamitos (2004)
9. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engeneering Methodologies 1(4), 355–398 (1992)
10. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2003)
11. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: controlling the scope of change in Java. In: OOPSLA 2005: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, And Applications, pp. 177–189. ACM Press, New York (2005)
12. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA 1990: Proceedings of the European Conference on Object Oriented Programming Systems Languages and Applications, pp. 303–311. ACM, New York (1990)
13. Costanza, P., Hirschfeld, R.: Language Constructs for Context-oriented Programming: An Overview of ContextL. In: Proceedings of the 2005 Symposium on Dynamic Languages, pp. 1–10. ACM Press, New York (2005)
14. Costanza, P., Hirschfeld, R., De Meuter, W.: Efficient Layer Activation for Switching Context-dependent Behavior. In: Lightfoot, D.E., Szyperski, C. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 84–103. Springer, Heidelberg (2006)
15. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A Mechanism for Fine-Grained Reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)
16. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp. 1–18. ACM Press, New York (2007)
17. Eugster, P., Jayaram, K.R.: EventJava: An Extension of Java for Event Correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)
18. Filman, R.E., Elrad, T., Clarke, S., Aksit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
19. Gonzlez, S., Mens, K., Cdiz, A.: Context-Oriented Programming with the Ambient Object System. Journal of Universal Computer Science 14(20), 3307–3332 (2008)
20. Hedin, G., Magnusson, E.: JastAdd: An Aspect-oriented Compiler Construction System. Science of Computer Programming 47(1), 37–58 (2003)
21. Hirschfeld, R., Costanza, P., Haupt, M.: An Introduction to Context-Oriented Programming with ContextS. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007 II. LNCS, vol. 5235, pp. 396–407. Springer, Heidelberg (2008)
22. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology 7(3), 125–151 (2008)
23. Hirschfeld, R., Perscheid, M., Schubert, C., Appeltauer, M.: Dynamic contract layers. In: 25th Symposium on Applied Computing, Lausanne, Switzerland. ACM DL, New York (2010)
24. Steele Jr., G.L.: Common LISP: The Language, 2nd edn. Digital Press, Newton (1990)

25. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
26. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
27. Núñez, A., Noyé, J., Gasiūnas, V.: Declarative Definition of Contexts with Polymorphic Events. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–6. ACM Press, New York (2009)
28. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
29. Schippers, H., Haupt, M., Hirschfeld, R., Janssens, D.: An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In: Proc. SAC PSC. ACM Press, New York (2009)
30. Smith, R.B., Ungar, D.: Programming as an experience: The inspiration for self. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 303–330. Springer, Heidelberg (1995)
31. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented Programming: Beyond Layers. In: Demeyer, S., Perrot, J.-F. (eds.) ICDL 2007: Proceedings of the 2007 International Conference on Dynamic Languages. ACM International Conference Proceeding Series, vol. 286, pp. 143–156. ACM Press, New York (2007)