Declarative Layer Composition in Framework-based Environments

Malte Appeltauer Robert Hirschfeld
Software Architecture Group
Hasso-Plattner-Institut, Germany
{first.last}@hpi.uni-potsdam.de

ABSTRACT

Context-oriented programming (COP) can improve modularity by dedicated language constructs for crosscutting concerns. Although COP could be used in any application domain in general, its current implementations may require adaptations of source code that is not accessible to the developer. This, in turn, limits the interaction of adaptation mechanisms provided by COP language extensions with widely used programming abstractions such as frameworks. As a result, dynamic control over layers emerges as a crosscutting concern that obstructs the separation of concerns.

In this paper, we discuss crosscutting layer composition in framework-based applications in detail. As a concrete example of such a framework-based application, we present a simple action adventure game that we implemented using a conventional COP language. Finally, we show, how our *JCop* language supports the modularization of these crosscutting concerns by language constructs for declarative layer composition.

Keywords

context-oriented programming, dynamic adaption, Java, framework

1. INTRODUCTION

The *context-oriented programming* (COP) [6, 4] approach supports the modularization of *crosscutting concerns* [11] and their control at runtime. In particular, COP focuses on a specific type of crosscutting concerns, the so called *heterogeneous crosscutting concerns* [1]. Heterogeneous crosscutting concerns require different source code to be executed at their *join points* (points in the program's structure or control flow [11]), whereas *homogeneous crosscutting concerns* require the same source code to be executed at their join points [1]. COP supports the implementation of such 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*. At runtime, the crosscutting behavior of layers can be composed with the core behavior of the classes. For this runtime layer composition, COP offers a block statement that declares a set of layers (specified by an argument list) and an execution scope (specified by the statement block) for which the layers should be composed with the base system. Hence, this construct scopes the effect of a layer composition to the dynamic extent of its statement block. Similarly, a second statement allows excluding layers from the execution. These two statements are typically denoted as `with` and `without` statements [6]. COP has been applied to several application domains where it showed to be a promising approach for the encapsulation of homogeneous crosscutting concerns.

However, research so far did not explicitly address the incorporation of COP with application domains that employ *frameworks* [8]. 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. A problem may occur, if a layer composition must be executed within the framework code, because one property that distinguishes frameworks from other libraries is that they prohibit access to their implementation [8]. Therefore, layer composition statements cannot be declared at source code locations in the framework (which would require an adaptation of the framework source code). 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 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 statement. The solution is to move the composition statement instead to a later point during the execution of the control flow at which 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 implementat 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

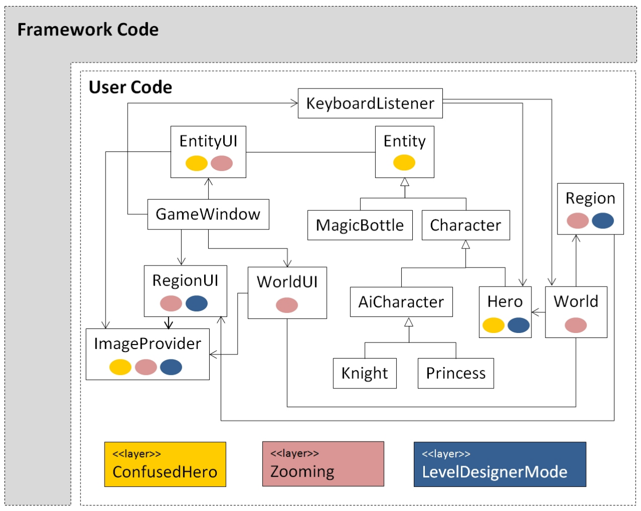

Figure 1: Class Diagram of the user code of RetroAdventure, where the colored circles represent layer adaptations.

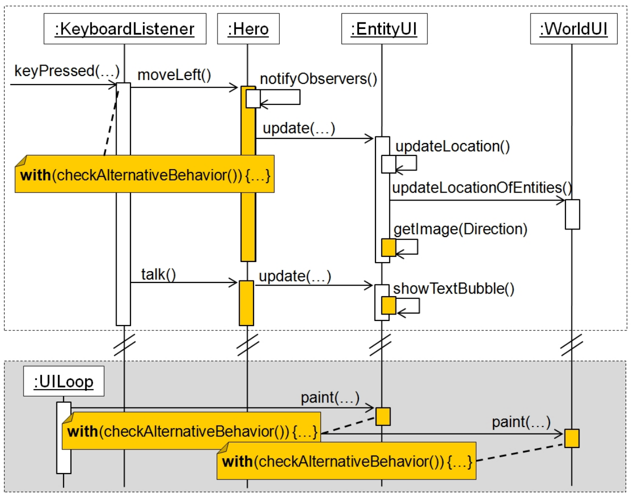

Figure 2: Redundant layer compositions in RetroAdventure(*gray:* framework thread, *white:* main thread).

composition logic. This crosscutting layer composition is a homogeneous crosscutting concern, which requires the same `with` statement to be repeated at multiple points.

In this paper, we discuss crosscutting layer composition in framework-based environments by example of a computer game that makes use of a graphical user interface (GUI) framework. For a better modularization, we propose to use the features of JCop that integrates COP with an *aspect-oriented programming* [11] language dedicated to the specification of declarative layer composition.

Section 2 gives an overview of our computer game example and explains how the GUI framework imposes a crosscutting implementation of layer composition. Section 3 shows how the aforementioned layer compositions in our example could be concisely expressed by JCop's language features. Section 4 reports on other case studies in which we identified that the use of frameworks causes crosscutting layer compositions. Section 5 discusses related work, and Section 6 concludes the paper.

2. CROSSCUTTING LAYER COMPOSITIONS IN FRAMEWORKS

In this section, we discuss the COP-based implementation of a simple action adventure game, which we implemented using our Java-based language *JCop* [2, 3]. JCop provides dedicated language constructs for COP, such as *layer* and *partial method* declarations, and *layer composition* statements. In addition, JCop integrates COP with a domain-specific aspect-language, see Section 3.

2.1 A GUI Framework-based Game

In the JCop-based *RetroAdventure* game, the user controls a hero character that moves through a world and can interact with items and computer controlled characters. The application employs the Swing [5] GUI framework for its graphics rendering. The user can move the hero through the world, let him speak to other characters, and collect items that are distributed in the world. In our object-oriented decomposition, which is shown in Figure 1, we identified several heterogeneous crosscutting concerns:

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 a period of time until the bottle magically fills up again. During that time, he cannot walk and speak properly, and the color of his face turns pink.

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

Graphics zooming The user can zoom in and out on the world, causing a scaling of the graphics and reloading of new images with higher level of detail.

We implemented these three crosscutting concerns by the layers `ConfusedHero`, `Zooming`, and `LevelDesignerMode`. In Figure 1, the classes that are adapted by a layer are marked with a circle in the same color as the layer.

2.2 Crosscutting Layer Composition

While the behavior of these three heterogeneous crosscutting concerns can be implemented by layers straightforwardly, we encountered an issue concerning the specification of their dynamic composition.

We explain that issue by the example of the layer `ConfusedHero` that implements the alternative hero 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(checkAlternativeBehavior()) { ... }
```

and the following auxiliary method[1]:

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

The `with` statement is used in the `keyPressed` method of the `KeyboardListener` class, so that any user-triggered control flow can activate the layer. Figure 2 (white box) presents a sequence diagram of that interaction. So far, we are able to concisely express the composition of `ConfusedHero` at only one point in our class decomposition.

However, during the execution of this user-triggered control flow, other threads may asynchronously call methods that are layered by `ConfusedHero` but without having this layer activated. For example, the Swing framework may asynchronously call the `paint` methods of the classes `EntityUI` and `WorldUI`. 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 2 (gray box) shows this framework-triggered control flow. Because we cannot access the source code of the UI loop[2] 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 initial intention of COP, which is the support of separation of concerns. The following section describes how crosscutting layer composition can be modularized using context classes in JCop.

3. SOLUTION: DECLARATIVE LAYER COMPOSITION

In this section, we present JCop's language features for declarative layer composition in framework-based environments. In most cases, context class declarations (Subsection 3.1) are the appropriate means to declaratively specify the source code locations at which the framework- and user-controlled threads must be adapted by a layer composition. In some cases, it is also of use to declare a layer to be globally active (Subsection 3.2), or let the layers themselves control their activation (Subsection 3.3).

[1]We assume, that the method `Bottle.isEmpty` returns `true` for a period of time after being drunk by the hero.

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

```
 1 public contextclass MagicBottleContext {
 2   public Bottle bottle;
 3   private Layer confusedBehavior = new ConfusedHero();
 4
 5   public MagicBottleContext(Bottle bottle) {
 6     this.bottle = bottle;
 7   }
 8
 9   public boolean heroDrunkTheBottle() {
10     return bottle.isEmpty();
11   }
12
13   when(heroDrunkTheBottle()) : with(confusedBehavior);
14 }
```

Listing 1: Context class declaration of the context-specific character behavior.

3.1 Context Classes

JCop's declarative layer compositions are encapsulated by a *context class declaration*, a special type declaration that can contain *pointcuts* and *advice* constructs, known form aspect languages such as *AspectJ* [10], and plain class members.

Listing 1 shows the implementation of the context class that controls the confused hero behavior in RetroAdventure. Its constructor is parameterized with a `Bottle` object and stores it in a class variable (Lines 5–7).

The actual layer composition is expressed in a declarative construct (Line 13). Syntactically, declarative compositions consist of two parts, a pointcut part and an advice part. The pointcut part is a logic expression consisting of built-in and named pointcuts. JCop's pointcut language consists of two main built-in pointcuts, `on` and `when`. The `on` pointcut describes methods whose entire execution should be adapted by a layer composition. The `when` pointcut describes a boolean expression that is evaluated at any point in the control flow at which a layer composition (described by the advice) may influence the execution [3]. The advice contains a sequence of with and/or a without operators.

The context class in Listing 1 uses a `when` pointcut to evaluate if the hero recently drunk the bottle. Therefore, a boolean method `heroDrunkTheBottle` checks if the bottle is empty (Lines 9–11). If true, then the advice activates an instance of `ConfusedHero`. The `when` pointcut evaluates its expression at the executions of any method that is adapted by a partial method of `ConfusedHero`.

A context class is instantiated like any Java class. Its layer composition can be dynamically deployed for the current thread by the `activate` method:

```
MagicBottleContext ctx = new MagicBottleContext(bottle);
ctx.activate();
```

Using this context class, we can concisely express the composition of `ConfusedHero` that was previously scattered over the object-oriented decomposition.

3.2 Static Active Layers

By default, layers in JCop are composed per control-flow. In addition, the layer declaration modifier `staticactive` de-

```
public staticactive layer Zooming {
  public BufferedImage gui.RegionUI.getClipForRegion()...
  public void gui.EntityUI.paint(Graphics g)...
  public Point gui.EntityUI.translate(Point local)...
  public Rectangle model.Region.getDimenson()...
  public Rectangle model.Region.getBorderBuffer()...
}
```

Listing 2: Top-level layer implementation of the graphics zooming concern.

clares that one *singleton* instance of the layer is implicitly globally activated on static initialization of the layer declaration. For the initialization of the singleton, the default constructor of the layer is used. This feature simplifies the declaration of crosscutting concerns that should be active during the entire execution of an application. In RetroAdventure, we use this modifier for the declaration of the layer `Zooming`, because the zooming feature is a static concern of the application, see Listing 2.

3.3 Layer-based 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 [2]. It gives access to inspect and manipulate layers, their composition and their partial methods at run-time.

In addition, JCop supports *layer-based composition* that allows layers themselves to manipulate layer compositions. This feature is implemented by an event handler mechanism. The event handler methods `onWith`, and `onWithout` are provided by the interface of `jcop.lang.Layer` (the implicit super type of all layers) and can be overwritten by concrete layer declarations. The handlers are implicitly called right after the execution of layer activations (`with`) and right before the execution of layer deactivations (`without`). The current composition is passed as an argument to the handler methods so that it can be analyzed and manipulated. The handler methods return a composition object that is activated instead of the input composition.

Listing 3 sketches the implementation of the layer `LevelDesignerMode` of RetroAdventure. Consider, we want to express that this layer should never be composed with `Zooming`. With the declaration of an `onWith` event handler (Lines 3–5), we can enforce this rule and implicitly deactivate any active instance of `Zooming` (using the `Composition.withoutAllLayers` method provided by JCop's reflective API).

4. FURTHER CASE STUDIES

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 and describe the results of our refactoring to JCop.

CJEdit [3, 2] is a simple editor that provides two modes of 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 activities, programming and documenting, require different functionality, which we implemented by layers. The text editor's core is implemented using the *Qt Jambi GUI Framework* [13].

```
public layer LevelDesignerMode {
  // composition handler
  public Composition onWith(Composition current) {
    return current.withoutAllLayers(Zooming.class);
  }
  // partial methods
  after public void gui.EntityUI.paint(Graphics g)...
  public void gui.RegionUI.paintComponent(Graphics g)...
  public BufferedImage gui.RegionUI.getImage()...
  public String[] gui.RegionUI.getInfo()...
}
```

Listing 3: A composition handler expresses the exclusion of `Zooming` for the activation of `LevelDesignerMode`.

WhenToDo [16] is a 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. WhenToDo uses a *context query framework* [16] that allows for reasoning about Web-based context information, and incorporates the reaction to context change with layer activation.

AstroPic [17] is an image gallery application for mobile devices. It automatically downloads and displays the current astronomy picture of the day with a short descriptive text. The application is implemented for the *Android* platform [14] 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.

YourBook 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 information such as age and visual defects. If, for example, a non-adult 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 YourBook Web shop is implemented using *Enterprise Java Beans* and the *JBossWS* Web service framework [15], which we extended to attach layer composition information to remote method calls.

Figure 3 gives an overview of these case studies. We first implemented the applications using our plain COP language *ContextJ* [2] and its `with` statement. As the table shows, layer composition is scattered over up to 33% of the user code classes. We then refactored the applications to JCop and used its declarative composition features. In all cases, layer composition could be fully encapsulated by context classes and other language features.

Case Studies	Project Size (User Code)		ContextJ-based Layer Composition		JCop-based Layer Composition	
	decomposition	lines of code	**implementation**	**tangled classes**	**implementation**	**tangled classes**
RetroAdventure Graphical Computer Game	33 classes 3 layers	3000	15 *with* statements	30%	2 context classes 1 static active layer	
CJEdit [4,5] GUI-based Text Editor	40 classes 6 layers	3500	13 *with* statements	13%	2 context classes 1 layer-based composition	
WhenToDo [20] Context-based ToDo App	25 classes 2 layers	1500	18 *with* statements	24%	1 context class context query library	**0%**
AstroPic [21] Mobile Image Gallery	9 classes 2 layers	320	3 *with* statements	33%	1 context class	
MyBook SOA-based Web Shop	30 classes 6 layers	4500	10 *with* statements	24%	1 context class JCop/SOA library	

Figure 3: Overview of the ContextJ/JCop implementations of layer composition in framework-based applications.

5. RELATED WORK

In this section, we discuss the language design and implementation of related COP languages with respect to their layer composition extensions.

Pointcut-based Declaration. The *EventCJ* [9] language has been published shortly after JCop [3] and both languages are closely related. Both languages are based on ContextJ and extend it with a domain-specific pointcut language for declarative layer activation. However, EventCJ and JCop use different built-in pointcuts and advice semantics. Like JCop, EventCJ uses pointcuts for layer activation. However, JCop restricts its join point model to method executions and dynamic conditions, whereas EventCJ inherits the whole AspectJ join point model [10].

Furthermore, layer composition within the advice is different in both languages. In EventCJ, it is defined by transition rules that can express conditional layer activation. JCop does not provide a dedicated syntax for such conditional layer activations. However, using JCop's reflective library and its ability to manipulate the layer composition, it is possible to provide the behavior of explicit syntax of EventCJ, though the EventCJ syntax is more concise and declarative.

Layer Guards and Implicit Layer Activation. In Subsection 3.3, we explained that in some scenarios layer composition requires the inclusion or exclusion of other layers, for which JCop provides layer activation handlers.

EventCJ also provides layer activation handlers that can execute additional functionality on layer composition. However, they cannot influence the layer composition.

The Python extension *ContextPy* [7] provides the concept of *guards* to declare layer relationships. Guards are functions that are assigned to a partial method. On method execution, they receive the list of currently active layers and return a Boolean value indicating whether the partial method the guard was assigned should be activated.

Similarly, another Python extension, *PyContext* [18], 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.

Object Structure-based Composition Scopes. The JavaScript extension *ContextJS* [12] 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. CONCLUSIONS

COP language extensions support the modularization of homogeneous crosscutting concerns. However, current COP implementations require access to the whole application source code to adapt it by layer composition statements. In many application domains, especially in framework-based environments, this requirement is not given, which leads to a crosscutting implementation of layer compositions.

We identified this problem in several applications and propose to address it by an aspect-oriented extension to current COP implementations that specifically serves for the declarative specification of layer composition. Our JCop programming language provides such an aspect language, in form of context classes, and integrates that feature with layers and partial methods. In addition, JCop supports to declare layers to be globally active, and its rich reflective API allows layers to reason about their composition. With these features, JCop's language constructs can help to enhance the separation of crosscutting concerns also in framework-based environments.

Acknowledgements

We would like to thank Patrick Henning, Daniel Moritz, Philipp Giese, and Bernhard Rabe for their contribution to the development of the JCop/SOA library and the YourBook application.

7. REFERENCES

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

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ - Context-oriented Programming for Java. *Computer Software of The Japan Society for Software Science and Technology*, 28(1):1_272–1_292, 2011.

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-based Software Composition in Context-oriented Programming. In *Proceedings of the 9th International Conference on Software Composition*, Lecture Notes in Computer Science, pages 50–65, Berlin, Heidelberg, Germany, 2010. Springer-Verlag.

[4] 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.

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

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

[7] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *25th Symposium on Applied Computing, Lausanne, Switzerland*, New York, NY, USA, 2010. ACM DL.

[8] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[9] Tetuso Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing Event-based Context Transition in Context-oriented Programming. In *Proceedings of The Second International Workshop on Context-Oriented Programming, COP'10*, pages 1–6, New York, NY, USA, 2010. ACM Press.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *The 15th European Conference on Object-Oriented Programming, ECOOP'01*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354, Berlin, Heidelberg, Germany, January 2001. Spinger-Verlag.

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

[12] 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, December 2011.

[13] Nokia Corporation. Qt 4.6 Whitepaper, 2009. http://qt.nokia.com/files/pdf/qt-4.6-whitepaper *(visited: 2011-12-09)*.

[14] Open Handset Alliance. Android Developers Platform. http://developer.android.com *(visited: 2011-12-09)*.

[15] RedHat inc. JBoss, 2011. http://www.jboss.com *(visited: 2011-12-19)*.

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

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

[18] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented Programming: Beyond Layers. In *Proceedings of the 2007 International Conference on Dynamic Languages, ICDL'07*, volume 286 of *ACM International Conference Proceeding Series*, pages 143–156, New York, NY, USA, 2007. ACM Press.