JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 3, March–Aprile 2008

Context-oriented Programming

Robert Hirschfeld,
Hasso-Plattner-Institut, Universität Potsdam, Germany
Pascal Costanza,
Programming Technology Lab, Vrije Universiteit Brussel, Belgium
Oscar Nierstrasz,
Software Composition Group, Universität Bern, Switzerland

Context-dependent behavior is becoming increasingly important for a wide range of application domains, from pervasive computing to common business applications. Unfortunately, mainstream programming languages do not provide mechanisms that enable software entities to adapt their behavior dynamically to the current execution context. This leads developers to adopt convoluted designs to achieve the necessary runtime flexibility. We propose a new programming technique called *Context-oriented Programming* (COP) which addresses this problem. COP treats context explicitly, and provides mechanisms to dynamically adapt behavior in reaction to changes in context, even after system deployment at runtime. In this paper, we lay the foundations of COP, show how dynamic layer activation enables multi-dimensional dispatch, illustrate the application of COP by examples in several language extensions, and demonstrate that COP is largely independent of other commitments to programming style.

1 INTRODUCTION

Contextual information is playing an increasingly important role for applications and services ranging from those that are location-based to those that are situation-dependent or even deeply personalized. While context-awareness is already an integral part of regular business applications, it is becoming even more critical for mobile and ubiquitous computing, where devices must adapt their behavior to the services available in their current environment.

Despite the fact that context is clearly a central notion to an emerging class of applications, there is little explicit support for context awareness in mainstream programming languages and runtime environments. This makes the development of these applications more complex than necessary. In this paper we argue the need for a new programming approach, called *Context-oriented Programming* (COP), which treats context explicitly and makes it accessible and manipulable by software.

One difficulty in proposing a concrete COP language is that *context* covers a wide range of concepts ranging from domain-specific to technology-dependent attributes, and including properties that may be spatial or temporal, or even based in hardware or software. We see personalization, sharing, location-awareness, ubiquitous computing, software evolution, runtime adaptation, and execution context dependencies as a part of COP's application domain.

At present, the lack of programming mechanisms to support the development of context-aware applications forces the design of these applications to be more complex and fragile than need be the case.

This paper presents Context-oriented Programming (COP) as a new programming technique to enable context-dependent computation. We claim that Context-oriented Programming brings a similar degree of dynamicity to the notion of behavioral variations that object-oriented programming brought to ad-hoc polymorphism. In support of this claim, we argue that the dynamic representation of layers and their scoped activation and deactivation in arbitrary places of the code are the essential ingredients for COP. Notably, in this paper we discuss:

- The motivation of Context-oriented Programming (COP) as a programming technique that directly supports context-dependent behavioral variations.
- Layers as named first-class entities that can be referred to explicitly at runtime, and whose composition can be dynamically controlled on-demand.
- An illustration of the utility of COP by several examples implemented in different COP extensions to Java, Squeak/Smalltalk and Common Lisp.

Specifically, we show as the contributions of this paper that:

- COP is independent from how source code is organized into textual modules.
- It can be beneficial to activate/deactivate layers from anywhere in the code.
- The notion of layered slots in ContextL allows a higher-order reflective programming style to integrate crosscutting concerns.

2 PROBLEMS

A context-dependent application varies its behavior according to conditions arising during execution. In order to pin down the term *context*, we offer the following abstract picture of context-dependence (see Figures 1a, b, and c).

An *actor* (as in UML) is an entity that interacts with the system, calling functions, sending messages, or employing other means of interaction in order to request the desired system behavior. A *system* is a computational entity that provides some desired behavior whenever requested. A system may be large or small, finely-grained or coarsely-grained. The constituents of a system may, for example, be procedures, methods, objects, components or subsystems. An *environment* represents anything external to the relationship and interaction between actor and system.

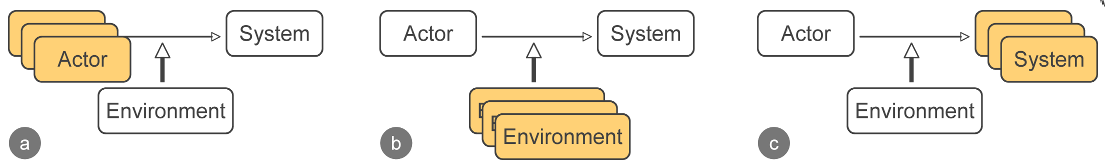

Figure 1: Actor-, environment, and system-dependent behavior variations.

The actors, the system, and the environment each contribute to the context that may influence the behavior of the application. Actor-dependent, system-dependent, and environment-dependent variations are described independently and separately, but can occur in any combination. Note that actor, system, and environment represent distinct *roles* that depend on a particular point of view at a specific point in time and can in principle be filled by all involved entities interchangeably. A piece of software may play the role of the system in one interaction, the actor in another, and provide the environmental context for yet a third. Examples of actors and systems are senders and receivers of messages in object systems, or clients and servers of remote procedures in distributed environments.

Actor-dependent Behavior Variations

An example of actor-dependent behavior variation is that of multiple visualizations of the same system or system entity, such as the rendering of statistical data as in different kinds of charts. Here, an actor determines which information provided by, or obtainable from, the system will be shown and in what form, contributing to the context-dependent behavior of the system.

A system needs to behave differently not only in response to different requests, but often also in response to the same request by different actors (Figure 1a).

Environment-dependent Behavior Variations

An environment-dependent behavior variation is essentially any conditional guarding a subset of application behavior or execution. Examples are anything that is not given implicitly by the flow of control in the execution of a piece of code but needs to be checked explicitly, such as the object a variable is referring to, the time of day, the battery status of the current device, or the temperature read out of sensory equipment. Often, related code is scattered over several system parts to coordinate activation or blocking of related environment-dependent behavior variations.

A system's response to a stimulus initiated by an actor may need to be adjusted to take properties of the computational environment into account. Here, the system's behavior can be affected by anything adjunct to the immediate interaction between actor and system (Figure 1b).

System-dependent Behavior Variations

An example of system-dependent behavior variation is that of change notification, the dissemination of information about what system parts have changed, and how. The way change notifications can be observed should depend on the relationships between the various system parts involved or affected. Often, multiple and redundant notifications are generated for a single change, resulting in cascading updates or other undesirable effects.

A system may need to vary its behavior depending on its own current state, historical information, or dependencies to other system parts or subsystems. In such case, the context that influences behavioral variation is not determined by different actors, but by the system itself (Figure 1c).

3 CONTEXT-ORIENTED PROGRAMMING

Context-oriented Programming enables the expression of behavioral variation dependent on context. To analyze and distill what constructs are necessary to enable expressing context-dependent behavioral variations, we have carried out a number of application and language extension experiments. Among others, we have implemented exception handling [2], multiple views on the same object which are selected based on execution context [14], coordination of screen updates [16], discerning of phone calls based on the context of both callers and callees [41], selecting billing schemes based on dynamic context conditions [15], and traversals of expression trees [26]. ContextL, our first language extension that explicitly supports our vision of Context-oriented Programming, has already been integrated into Lisp on Lines, a Web framework that is used in commercial applications [12], and is used for generating different document formats (like html, pdf, etc.) from the same object structures. In other settings, we have developed Piccola, a language with first-class namespaces [1], a context-oriented extension of AspectS [25] for Smalltalk/Squeak, ContextS for Smalltalk/Squeak, and context-oriented extensions of AmbientTalk [42].

Based on these implementations of both application scenarios and necessary language extensions, we identify the following essential language properties to support COP: (a) means to specify *behavioral variations*, (b) means to group variations into *layers*, (c) dynamic *activation* and *deactivation* of layers based on *context*, and (d) means to explicitly and dynamically control the *scope* of layers. Based on our findings, approaches to COP should at least address the following properties:

Behavioral variations. Variations typically consist of new or modified behavior, but may also comprise removed behavior. They can be expressed as partial definitions of modules in the underlying programming model such as procedures or classes, with complete definitions representing just a special case. Variations may also be expressed as edits, wrappers, or even general refactorings or transformations.

Layers. Layers group related context-dependent behavioral variations. Layers are first-class entities, so that they can be explicitly referred to in the underlying programming model.

Activation. Layers aggregating context-dependent behavioral variations can be activated and deactivated dynamically at runtime. Code can decide to enable or disable layers of aggregate behavioral variations based on the current context.

Context. Any information which is computationally accessible may form part of the context upon which behavioral variations depend.

Scoping. The scope within which layers are activated or deactivated can be controlled explicitly. The same variations may be simultaneously active or not within different scopes of the same running application.

Layers are composed in reaction to contextual information. Based on information available in the current execution context, specific layers may be activated or deactivated. COP languages and environment extensions provide the mechanisms for expressing, activating and composing layers at runtime, but it is the application domain that determines which contextual information is relevant. This last part may be counter-intuitive, but based on our experience, it is indeed not necessary to provide explicit support for context modeling, since existing object-oriented abstraction mechanisms are sufficient to model context [17].

4 MULTI-DIMENSIONAL MESSAGE DISPATCH

Following Smith and Ungar [38], we present COP in the context of multi-dimensional message dispatch. While this is not to suggest that COP implementations have to be based on such concepts, we believe that this analogy will help the reader to understand that COP is a continuation of other work, namely procedural, object-oriented, and subjective programming.

One-dimensional dispatch. Procedural programming provides only one dimension to associate a computational unit with a name [38]. Here, procedure calls or names are directly mapped to procedure implementations. In Figure 2a calling $\mathtt{m}_1$ leaves no choice but the invocation of the only implementation of procedure $\mathtt{m}_1$.

Two-dimensional dispatch. Object-oriented programming adds another dimension for name resolution to that of procedural programming [38]. In addition to the method or procedure name, message dispatch takes the message receiver into consideration when looking up a method. In Figure 2b we see two implementations of method $\mathtt{m}_1$. The selection of the appropriate method not only depends on the message name $\mathtt{m}_1$, but also the receiver of the actual message, here $\mathtt{R}_Y$.

Three-dimensional dispatch. Subjective programming as introduced by Smith and Ungar [38] extends object-oriented method dispatch by yet another dimension.

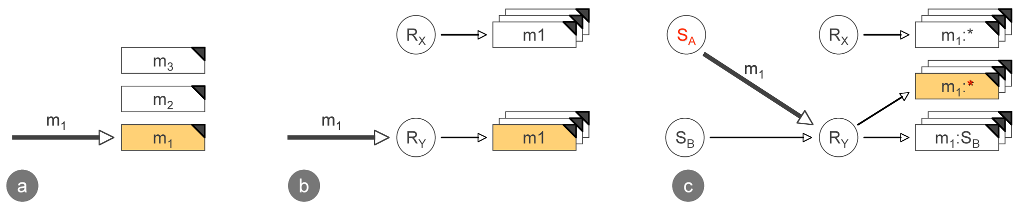

Figure 2: One-, two-, and three-dimensional method dispatch.

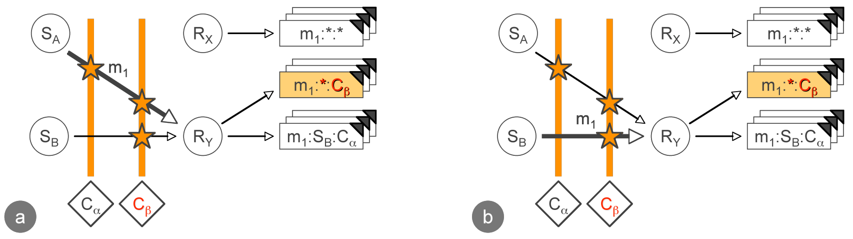

Figure 3: Four-dimensional method dispatch.

Here, methods are not only selected based on the name of a message and its receiver but also on its sender. In Figure 2c sender S_A sends message m_1 to receiver R_Y. R_Y implements two methods m_1: The first one is associated with any sender ($m_1:*$), but the second one only with sender S_B ($m_1:S_B$). This is why, in our example, $m_1:*$ is selected for execution.

Four-dimensional dispatch. Context-oriented Programming takes subjective programming one step further by dispatching not only on the name of a message, its sender and its receiver, but also on the context of the actual message send. Context, as stated in Section 3, is anything that is computationally accessible, either directly or derived. Based on context information, methods or their partial definitions are selected for or excluded from message dispatch, leading to context-dependent behavioral variations as described in the previous sections.

In Figure 3 there are two scenarios showing how context can affect method dispatch in COP environments. In both scenarios a message m_1 is sent to the same receiver R_Y, from different senders (S_A and S_B), and in different contexts (C_α and C_β). In Figure 3a, our selection process results in method $m_1:*:C_\beta$. This is because that particular partial method implementation matches with messages m_1, sent to receivers R_Y, by any sender ($*$ is matched by sender S_A), in both contexts C_α or C_β. In Figure 3b, our selection process results also in method $m_1:*:C_\beta$, now because the message and its sender and receiver correspond to the method's properties as before, and in addition to that, context C_β matches with m_1's C_β property as well. Method $m_1:*:C_\alpha$ will not match with our request due to incompatible context properties.

Please note that combinations of context-conditions can be considered in method dispatch but are not discussed in this paper.

```
class Person {
  private String name, address;
  private Employer employer;

  Person(String newName,
         String newAddress,
         Employer newEmployer) {
    this.name = newName;
    this.employer = newEmployer;
    this.address = newAddress;
  }

  String toString() {return "Name: "+name;}

  layer Address {
    String toString() {
      return proceed()+"; Address: "+address;
    }
  }

  layer Employment {
    String toString() {
      return proceed()+"; [Employer] "+employer;
    }
  }
}
```

```
class Employer {
  private String name, address;

  Employer(String newName,
           Stringe newAddress) {
    this.name = newName;
    this.employer = newEmployer;
  }

 String toString() {return "Name: "+name;}

 layer Address {
   String toString() {
     return proceed()+"; Address: "+address;
   }
 }
}
```

Figure 4: Layers providing different display behavior.

5 EXAMPLES

In this section, we come back to the problems described in Section 2 and use them to both describe COP-based example solutions as well as language extensions. We first discuss actor-dependent behavior variations using ContextJ for Java to implement multiple views on objects. Then, we show how to conditionally activate and deactivate behavioral variations based on explicitly checking properties that are not directly associated with the current flow of control using ContextS for Squeak/Smalltalk. Finally, we illustrate system-dependent behavior variations based on an implementation of change notifications in ContextL for Lisp.

In addition to illustrating how COP can be used to address context changes caused by actors, the environment, and the system itself, our examples also show how behavorial variations can be expressed in different ways. While examples discussed in our previous publications about COP [14, 16] are expressed in a style close to aspect- and feature-oriented programming, we deviate here in several ways. Specifically, we show that (a) placing the behavioral variations associated with layers in the respective class definitions instead of textually modularizing layers is a valid organization of the source code for COP, (b) there is no need to expose AOP-style join points to enable the use of layer activation and deactivation, and (c) the notion of context-dependent behavioral variations expressed as layers is compatible with a higher-order reflective programming style to handle crosscutting concerns.

Actor-dependent Behavior Variations

The following example is expressed in ContextJ. We consider ContextJ to be very helpful in illustrating COP language constructs to programmers more fluent in mainstream programming languages than in Lisp or Smalltalk. A different variant of ContextJ was already used as a means to ease the accessibility of a ContextL-based example [16].[1]

The code in Figure 4 is a simplified version of an earlier example [14]. Here, we focus on elements essential to this paper showing a ContextJ implementation of actor-dependent behavior variations where the views a system presents to its clients depend on the clients requesting such views.

In Figure 4, we define the classes `Person` and `Employer` with fields `name`, `address` and `employer`, together with the necessary constructors and a default `toString` method. We also define the two layers `Address` and `Employment` that define behavioral variations on `toString`. In the `Address` layer, address information is returned for instances of `Person` and `Employer` in addition to the default behavior of `toString`. The call to the special method `proceed` ensures that the original definition of `toString` is called. In this sense, `proceed` is similar to `super` in Java for performing supercalls, or `proceed` in AspectJ for calling original method definitions from around advice. The `toString` method in layer `Employment` returns additional information about the employer of a person in the `Person` class.

None of the user-defined layers `Address` and `Employment` are activated by default. Instead, a client program must explicitly choose to activate them when desired. ContextJ provides `with` and `without` for activation and deactivation of layers with dynamic scope.

Here, the purpose is to present different views on the same program where each client can decide to have access to just the name of persons, their employment status, or the addresses of persons, or employers, or both. For example, when a client chooses to activate the `Address` layer but not the `Employment` layer, address information of persons will be printed in addition to their names. When the `Employment` layer is activated on top, a request for displaying a person object will result in printing that person's name, its address, its employer, and its employer's address, in that particular order. A code fragment showing the activation of these two layers is given in Figure 5.

Note that layer activation affects the behavior of the entire program in the dynamic extent of the `with` construct, not on selected objects. Furthermore, the code in Figure 4 duplicates major parts of the code for classes `Person` and `Employer`. However, this is for illustration purposes only since both classes can as well inherit a common code base from a shared superclass, including shared implementations for the different layers.

[1]Please note that ContextJ is not yet implemented. In Appendix A we provide a library-based implementation of a subset of ContextJ called ContextJ$\star$.

```
Employer vub      = new Employer("VUB", "1050 Brussel");
Person somePerson = new Person("Pascal Costanza", "1000 Brussel", vub);

with (Address) {
  with (Employment) {
    System.out.println(somePerson);
  }
}

Output:  Name: Pascal Costanza; Address: 1000 Brussel;
         [Employer] Name: VUB; Address: 1050 Brussel
```

Figure 5: Direct layer activation.

Contrary to the examples discussed in previous publications about our work on COP, this code example does not modularize the source code along the layers, but keeps the object-oriented modularization along classes. In other words, instead of textually grouping partial class definitions inside layers – which is similar to grouping aspects or features within their own modules in AOP and FOP – the layer definitions are textually grouped inside classes. This layer-in-class notation has different trade-offs than the class-in-layer notation. One advantage is that layers can refer to private fields of the core class definition, whereas in class-in-layer notations, members that need to be referred to in other layers (aspects, features, etc.) cannot be fully encapsulated. Another advantage is that new classes can be added to a system offering their own layer-specific behavior without the need to change layer definitions elsewhere.

Our example, as discussed here, uses simple print statements to display information about objects, which in turn call the `toString` methods. In situations with more advanced GUI presentations required, state changes need to initiate automatic updates of such on-display presentations. In Section 5, we show how COP can be beneficial in the implementation of display update behavior.

Environment-dependent Behavior Variations

In the previous example, the composition of behavioral variations is explicitly initiated by actors using the `with` construct from within the control flow of the program. In the general case, however, activation or deactivation of layers may depend on non-local contextual conditions of the environment. For example, whether address information (or other kinds of information) is displayed may depend on the setting of some menu entry in the client program, on a user's login status, whether a user has sufficient access rights to view sensitive information, and so on. In general, any dynamic condition can determine whether some layer should be activated or not.

The `with` construct is in principle already sufficient to express this: It can just be guarded by an appropriate `if` statement. In Figure 6, this is shown for both ContextJ and ContextS, our COP extension to Squeak/Smalltalk. The disadvantage

```
-[1]- Direct activation with code replication
      (Java/ContextJ)

if (currentUser().verboseMode())
  with (Address) {
    for (Person p: personList)
      System.out.println(p);
  }
else for (Person p: personList)
       System.out.println(p);

-[2]- Direct activation with code replication
      (Smalltalk/ContextS)

currentUser verboseMode
  ifTrue: [
    [ { AddressLayer new. } ]
      useAsLayersFor: [
        personList do: [:each |
          Transcript cr; show: each printString]]]
  ifFalse: [
    personList do: [:each |
      Transcript cr; show: each printString]]
```

```
-[3]- Direct activation without code replication
      (Smalltalk/ContextS)

(currentUser verboseMode
   ifTrue: [ { AddressLayer new. } ]
   ifFalse: [])
     useAsLayersFor: [
       personList do: [:each |
         Transcript cr; show: each printString]]

-[4]- Derived activation based on context object
      (Smalltalk/ContextS)

context computeLayers
  useAsLayersFor: [
    personList do: [:each |
      Transcript cr; show: each printString]]
```

Figure 6: Conditional layer activation.

in this case, though, is that the code on which layers may need to be activated or not must be duplicated for each branch (code sections 1 and 2 in Figure 6).

In ContextS we implemented an alternative approach to conditional layer activation: The message `useAsLayersFor:` can be sent to a sequenceable collection that contains the layers to be activated when evaluating the block passed as argument to `useAsLayersFor:`. This property can be used to avoid code duplication in the alternative branches of a guarding clause (code section 3 in Figure 6).

In principle, ContextJ can be designed to support first-class layers and computed layer activation as well (see Appendix A). The solution in ContextS is remarkable, though, in that Smalltalk lends itself to designing layers as first-class objects from the start and that especially Smalltalk's elegant block construct makes syntactic language extensions for layer activation and deactivation unnecessary.

This allows us to avoid explicit conditionals in general, and to use dedicated computations that derive layer combinations appropriate for a particular execution context. With that, context can be modelled as first-class objects that, for example, can be used as shown in section 4 of Figure 6.

System-dependent Behavior Variations

To illustrate a COP-style implementation of system-dependent behavior variations, we show a modified version of the figure editor code written in ContextJ [16], here implemented in ContextL, our first fully implemented programming language extension for COP [14] built on top of the Common Lisp Object System (CLOS). The figure editor is a prominent example used in aspect-oriented software development to motivate cflow-like constructs needed to deal with the phenomenon of jumping aspects [11].

```
;;; Abstract figure elements

(define-layered-class figure-element)

;;; Display Layer

(deflayer display)

(defmethod call&update ((this figure-element) change-function)
  (let ((result (with-inactive-layers (display)
                  (invoke change-function))))
    (update-display this)
    (return result)))

(define-layered-method layered-slot-set :in-layer display :around ((this figure-element) writer)
  (call&update this writer))

(define-layered-method move :in-layer display :around ((this figure-element) dx dy)
  (call&update this (function call-next-method)))

;;; Points and Lines

(define-layered-class point (figure-element)
  ((x :initarg :x :layered t :accessor point-x)
   (y :initarg :y :layered t :accessor point-y)))

(define-layered-method move ((this point) dx dy)
  (setf (point-x this) (+ (point-x this) dx))
  (setf (point-y this) (+ (point-y this) dy)))

(define-layered-class line (figure-element)
  ((p1 :initarg :p1 :layered t :accessor line-p1)
   (p2 :initarg :p2 :layered t :accessor line-p2)))

(define-layered-method move ((this line) dx dy)
  (move (line-p1 this) dx dy)
  (move (line-p2 this) dx dy))
```

Figure 7: A layered implementation of the figure editor example.

In Figure 7, we see a layered class `figure-element` providing a simplified abstract interface to graphical objects to be presented on the screen. The code also defines a layer `display` that handles display updating on slot changes: Whenever a *layered* slot is changed through an assignment, the reflective function `layered-slot-set` is called on the object whose slot is to be assigned, together with a first-class function `writer` that performs the actual assignment when invoked. The function `layered-slot-set` is overriden for `figure-element` to perform the actual update of grapical objects on the screen (assumed to be implemented by the function `update-display`, which is not detailed further in this paper).

Figure 7 also contains two example classes `point` and `line` that inherit from `figure-element`. They define x- and y-coordinates, and two endpoints respectively, together with specifications for implicit initialization arguments for constructors `:initarg` and accessor methods `:accessor`, which are standard slot options in CLOS. ContextL adds the `:layered` option that indicates whether the respective slot is layered or not. This allows fine-grained control over which slots are read and written through reflective functions and allows expressing, in this example, whether a change to a specific slot should trigger an update to the screen or not.

In the general case, graphical objects may be either primitive, like `point`, or composites, like `line`, which consists of two endpoints. Usually every change to a composite object, for example by calling `move` on a `line` instance, implies respective changes to the primitive objects the composite combines. However, a call of a top-level `move` should only cause a single notification after performing all implied changes to the combined objects to avoid unnecessary and inaccurate intermediate screen updates. To this end, both `layered-slot-set` and `move` are defined in the layer `display` to first deactivate `display` itself for the dynamic scope of the actual change of the figure element. In order to avoid code duplication, the common code for `layered-slot-set` and `move` is factored out into a higher-order function `call&update` that expects a parameterless function for performing the update on an object and the object itself. The around method for `layered-slot-set` in the layer `display` passes the `writer` parameter, and the around method for `move` in `display` passes the CLOS construct for super calls, `call-next-method`, as a function.

It is this deactivation of the layer `display` in `call&update` that is neither triggered by an actor nor by dynamic conditions in the environment, but is internally required by the system to coordinate display updating.

Discussion

We have already introduced ContextL and a variation of ContextJ in previous publications on COP. The examples given above summarize the examples discussed before to illustrate COP as a more general concept, but also discuss new elements of the respective languages. Notably: (a) We have shown ContextJ with an AOP/FOP-style class-in-layer notation before, but this paper introduces the layer-in-class notation for the first time. (b) ContextS is used here to illustrate an elegant way to deal with layers as first-class objects. Layers as first-class objects are also available in ContextL but have not been discussed before. (c) In the ContextL example shown here, we discuss layered slots for the first time which allow for intercession of slot accesses in layered methods.

These examples all show that the notion of layer activation and deactivation is a useful concept of its own, independent of how the source code is structured to define layer-specific behavior. Indeed, the essential contribution of COP is the ability to refer to layers at runtime and activate or deactivate them in arbitrary code locations. In other words, COP brings a similar degree of dynamicity to the notion of crosscutting concerns that object-oriented programming brought to ad-hoc polymorphism.

```
for (Person p: personList) {
  System.out.println(p);
  with (Address) {System.out.println(p.getEmployer());}
}
```

Figure 8: Layer activation from within a loop.

In this regard, it is especially interesting to compare COP to cflow-style constructs from aspect-oriented programming. Cflow constructs, as introduced with AspectJ-like languages, match join points within the dynamic extent of another join point [29]. They allow for the expression of control-flow-dependent behavior variations and, in conjunction with `if` pointcuts, the conditional composition of such behavior variations. In this sense, AOP languages with cflow constructs can already be regarded as supporting a context-oriented programming style. However, AOP has emerged from the ambition to textually modularize crosscutting concerns. In this sense, the layer-in-class notation used in the ContextJ example above would not qualify as AOP because it scatters layer-specific definitions in the classes to which they belong.

Another important difference between cflow-style pointcuts in AOP and layer activation/deactivation in COP is that COP allows the use of layer activation/deactivation in arbitrary places of the source code whereas pointcuts in AOP can only trigger at join points as exposed by the rest of the program. For example, consider the ContextJ example from above and assume that we want to print a list of persons with their employers and the employers' addresses, but not addresses of the persons themselves. In a COP style, we can write this down directly (Figure 8).

With cflow-style pointcuts, we would have to ensure that the relevant part of the loop body is exposed as join points, for example by factoring it out into a separate method or using statement annotations [18]. In both cases, the join points would have to be exposed, which means that encapsulation boundaries are potentially violated and that the join point interface has to be maintained for future versions of the code due to the fragile pointcut problem [31].

In the figure editor example, a reflective hook is used for intercepting slot accesses. This is modelled along the lines of, and actually in terms of, the similar slot access protocol in the CLOS Metaobject Protocol [28]. This style is a natural fit for the typical higher-order programming styles which are traditionally used in Lisp dialects to integrate crosscutting and non-functional concerns. This example is shown here to illustrate further that the notion of dynamic layer activation and deactivation is independent of other elements of programming style supported by a given language.

Furthermore, the conditions for triggering updates in the figure editor example continuously change at runtime. This requires efficient implementations from both aspect-oriented-style cflow constructs as well as context-oriented-style layer activation and deactivation. We have previously shown how ContextL can be implemented efficiently [16]. In aspect-oriented approaches, both static analysis [5] and guards [10] show substantial improvements over naive checks of control flow properties. The latter implementation strategy, as used in the newest version of the Steamloom virtual machine, is a dynamic optimization and therefore especially interesting for possible future COP implementations.

6 RELATED WORK

Aspect-oriented and Feature-oriented Programming Aspect-oriented programming language extensions provide constructs for modularizing crosscutting concerns [30]. A crosscutting concern is some program behavior, be it functional or non-functional, that does not fit the dominant modularization of a program. For example in object-oriented languages, the dominant modularization is based on a class hierarchy where the only way to share state and behavior across several classes is by way of inheritance. An aspect can define additional state and behavior across several classes independent of the structure of the class hierarchy. An especially interesting contribution of aspect-oriented programming are means to decrease code scattering: The source code of a program exhibits code scattering when the same, or similar, fragments of code are repeated throughout. Aspects can help to reduce code scattering by abstracting away the places in a program where some code has to be executed, and defining that code only once and in a single location.

Like AOP, feature-oriented programming [7] also targets crosscutting concerns. For example, the mixin layers approach [37] models features as layers which consist of partial class definitions, similar to the layers of Context-oriented Programming as described in this paper. The approach taken in [4] is based on mixin layers that may include constructs for defining AOP-style pointcuts and advice, showing that AOP and FOP are not fundamentally incompatible. However, the focus of FOP is on compile-time selection and combination of feature variations, and more recent tool support incorporates algebraic means for reasoning about such feature combinations to improve static analyzability [6].

Context-oriented Programming, as presented in this paper, is closer to FOP than to AOP: It also takes the notion of layers and provides means for selecting and combining them. However, the main difference between FOP and COP is that the former focuses on compile-time selection and combination, whereas the latter adds the notion of dynamic activation and deactivation of layers together with scoping mechanisms to delimit their activity state.

Unlike AOP and FOP, COP does not require the textual modularization of crosscutting concerns in the source code of a program. As illustrated in Section 5, the behavioral variations may be placed in the source code by making them part of the respective class definitions. However, it must be possible to unambiguously refer to all the behavioral variations that belong to one layer. Otherwise, dynamic activation/deactivation of behavioral variations would not be feasible in a straightforward way. It is exactly this possibility to refer to crosscutting behavioral variations grouped as layers at runtime that is important for Context-oriented Programming.

Context-awareness The term Context-oriented Programming was first used to refer to the notion of context-aware applications in pervasive and ubiquitous computing. Early approaches focused on the modeling and provision of context information [17]. While this is an important prerequisite for implementing context-aware applications, Context-oriented Programming is concerned with programming language constructs to represent and manipulate behavioral variations. The first precursors to the generalized notion of Context-oriented Programming presented in this paper were placed in the context of pervasive and ubiquitous computing [20, 27].

Dynamic Activation Programming languages that treat procedures or methods as first-class values allow programs to be structured in a way that they can exhibit different behavior under different circumstances. In pure functional programming languages like Haskell, they can be passed as parameters to higher-order functions, and in languages like Scheme or ML, they can be assigned to variables which can be side-effected later on to change the behavior of a program. However, the availability of first-class procedures alone does not offer sufficient means to coordinate changes of several collaborating procedures.

First-class environments offer a way to group several variable bindings, which can contain first-class procedures or functions as values, and allow code to be executed in such environments or even manipulate their bindings. However, aside from some implementations of Scheme and Lisp, essentially no mainstream programming language provides first-class environments as a mechanism available to programmers.

Piccola [1] is an experimental language for specifying applications as compositions of software components. A key feature of Piccola is the *form* – a first-class environment which is used to model, amongst others, objects, components, modules and dynamic contexts [2]. Although environments can be manipulated in Piccola, expressions are statically bound to the environment they are evaluated in, so dynamic activation of scopes is not directly supported.

Scoped Activation Dynamic scoping can be used to change the binding of variables for the current execution context [22, 32, 39]. In languages with first-class functions, the change of function bindings can also be dynamically scoped, but special care has to be taken when different functional values for the same dynamically scoped variable need to wrap each other [13]. Dynamically scoped functions, however, still have the limitation that they do not offer explicit means to coordinate the change of collaborating functions. An interesting application of scoped activation of functions is described by Gabriel [19]. Local virtual functions, implemented as a pre-processor for C++, provide a similar concept [23].

Several aspect-oriented approaches already provide scoped aspect activation as abstractions [9, 24, 35, 40, 43], but these abstractions have foremostly been designed for aspect deployment. They are all suitable for Context-oriented Programming to different extents. Recently, ContextL [16] and Steamloom [10] have shown that scoped aspect/layer activation can be efficiently implemented, an important prerequisite for wider adoption. This paper shows that scoped layer activation is a language construct that is independent from the textual organization of source code, be it into aspect-oriented modules, feature-oriented layers or otherwise.

Other Related Work Context-oriented Programming combines layers, dynamic activation based on context, and scoped activation. Different combinations of these notions already exist, some of which are even quite old. The separation of programs into layers is an idea that goes back at least to early experiments made in Smalltalk [8, 21]. There, layers were conceived as a mechanism to express different design alternatives of the same software, and allow the developer to try different combinations thereof. Us [38], Delegation layers [34] and Slate [36] are very close to COP in that they are based on layers, allow sending messages in the scope of a specific layer, and even allow manual dynamic composition of layers. The context-oriented approaches presented in this paper abstract away from the need to explicitly compose layers and provide dedicated language constructs instead – `with-active-layers` and `with-inactive-layers` in ContextL, `with` and `without` in ContextJ, and `useAsLayersFor:` in ContextS. These language abstractions enable non-trivial and efficient implementations of the respective language extensions [15, 16].

Changeboxes [33, 44] offer a further evolution of the notion of Classboxes. A Changebox encapsulates a software change at a higher-level of abstraction than Classboxes (which only record additions, deletions or modifications to methods). Changeboxes can be composed to provide a runtime environment for software applications. Different Changeboxes can be in effect at the same time within a running application, and may be dynamically activated or deactivated, thus supporting a form of COP. In the present prototype, activation is specified programmatically, but implicit activation based on context is planned in the longer term.

7 OUTLOOK

Real software systems change over time, and must adapt to changing requirements even while they are running. Unfortunately, mainstream programming languages and development environments do not support this kind of dynamic change very well, leading developers to implement complex designs that anticipate various dimensions of variability.

We propose instead to develop a notion of Context-oriented Programming that directly supports variability depending on a large range of dynamic attributes. In effect, runtime behavior can be dispatched on any properties of the execution context. Our first prototypes have illustrated how COP supports multi-dimensional dispatch to achieve expressive runtime variation in behavior.

More research and experimentation is clearly needed into different ways to compose and trigger variations. For example, variations may be expressed not only as differences in code, but as extensions, wrappers, advice, refactorings, or general transformations. Variations may not only be triggered explicitly and programmatically, but also implicitly by means of rules or constraints.

One compelling application domain for COP is to control software evolution. COP can enable runtime deployment of software changes while strictly delimiting the visibility of changes to specific clients and leaving others untouched. Radical changes can slowly percolate through a running system, without breaking clients which are not ready for them. Backwards-compatibility need no longer be a critical concern if the client context is taken into account when deploying changes. Global consistency is neither assumed nor desirable.

COP signals a move away from application-specific solutions to runtime variability (such as those offered by many design patterns), and instead focuses on developing a new programming paradigm and corresponding language constructs and underlying software infrastructure to support context-dependent behavior. We believe that these steps are essential to achieving the high degree of maintainability, robustness and adaptability that is needed for the emerging breed of dynamic, mobile and pervasive applications.

Implementations of ContextJ⋆, ContextS, and ContextL can be downloaded from http://www.swa.hpi.uni-potsdam.de/cop/

Acknowledgements

We thank Sven Apel, Nick Bourner, Johan Brichau, Thomas F. Burdick, Drew Crampsie, Marcus Denker, Brecht Desmet, Stéphane Ducasse, Johan Fabry, Richard P. Gabriel, Michael Haupt, Christian Heinlein, Andy Kellens, Bjoern Lindberg, Martin von Löwis, Wolfgang De Meuter, Jim Newton, Andreas Raab, Andreas Rasche, Christophe Rhodes, Dave Thomas, Jorge Vallejos, Matthias Wagner, Peter J. Wasilko, JonL White, and Roel Wuyts for fruitful discussions and valuable contributions.

REFERENCES

[1] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola — a Small Composition Language. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[2] F. Achermann and O. Nierstrasz. Explicit Namespaces. In J. Gutknecht and W. Weck, editors, *Modular Programning Languages, Proceedings of JMLC 2000 (Joint Modular Languages Conference)*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, Sept. 2000. Springer-Verlag.

[3] M. Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*. ACM Press, Mar. 2003.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering 2006*, May 2006.

[5] P. Avgustinov, J. Tibble, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, and G. Sittampalam. Optimizing AspectJ. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM Press, 2005.

[6] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the International Conference on Software Engineering 2004*, 2004.

[7] D. Batory and A. Rauschmeyer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, June 2004.

[8] D. G. Bobrow and I. P. Goldstein. Representing design alternatives. In *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*, 1980.

[9] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[10] C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient control flow quantification. In *OOPSLA 2006, Proceedings*. ACM Press, 2006.

[11] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. In C. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr, editors, *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[12] B. Clementson. Vancouver Lisp Users Group meeting for December 13, 2005, http://bc.tech.coop/blog/051213.html.

[13] P. Costanza. Dynamically Scoped Functions. *SIGPLAN Notices*, 38(8):29–36, Aug. 2003.

[14] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming — An Overview of ContextL. In R. Wuyts, editor, *Proceedings of the 2005 Dynamic Languages Symposium*. ACM Press, Oct. 2005.

[15] P. Costanza and R. Hirschfeld. Reflective layer activation in ContextL. In *Proceedings of the Programming for Separation of Concerns (PSC) of the ACM Symposium on Applied Computing (SAC)*, LNCS. Springer Verlag, March 2007.

[16] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient layer activation for switching context-dependent behavior. In D. Lightfoot and C. Szyperski, editors, *Joint Modular Languages Conference 2006*. Springer LNCS 4228, Sept. 2006.

[17] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[18] M. Eaddy and A. Aho. Statement annotations for fine-grained advising. In *Proceedings of the ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, 2006.

[19] R. P. Gabriel. The design of parallel programming languages. *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Aug. 1991.

[20] M. L. Gassanenko. Context-oriented programming. In *euroForth'98*, Schloss Dagstuhl, Germany, April 1998.

[21] I. P. Goldstein and D. G. Bobrow. Extending object oriented programming in Smalltalk. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages 75–81. ACM Press, 1980.

[22] D. R. Hanson and T. A. Proebsting. Dynamic variables. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 264–273. ACM Press, 2001.

[23] C. Heinlein. Global and Local Virtual Functions in C++. *Journal of Object Technology*, 4(10):71–93, December 2005.

[24] R. Hirschfeld. AspectS – aspect-oriented programming with Squeak. In M. Akşit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[25] R. Hirschfeld and P. Costanza. Extending advice activation in AspectS. In *Proceedings of the AOSD Workshop on Open and Dynamic Aspect Languages (ODAL)*, 2006.

[26] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-oriented Programming with ContextS (to appear). In *2nd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, 2008.

[27] R. Keays and A. Rakotonirainy. Context-oriented programming. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 9–16, New York, NY, USA, 2003. ACM Press.

[28] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[29] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, volume LNCS 2072, pages 327–353, Berlin, June 2001. Springer.

[30] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conference Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[31] C. Koppen and M. Störzer. PCDiff: Attacking the fragile pointcut problem. In K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, Sept. 2004.

[32] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–118. ACM Press, 2000.

[33] O. Nierstrasz, M. Denker, T. Gîrba, and A. Lienhard. Analyzing, capturing and taming software change. In *In Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[34] K. Ostermann. Dynamically composable collaborations with delegation layers. In B. Magnusson, editor, *Proc. ECOOP 2002*, pages 89–110. Springer Verlag, June 2002.

[35] K. Ostermann and M. Mezini. Conquering aspects with Caesar. In Akşit [3], pages 90–99.

[36] L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*. Springer LNCS, 2005.

[37] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2):215–255, Apr. 2002.

[38] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[39] G. L. Steele, Jr. *Common Lisp the Language, 2nd Edition.* Digital Press, 1990.

[40] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. 23rd Int'l Conf. Software Engineering (ICSE'2001)*, May 2001.

[41] J. Vallejos, P. Ebraert, and B. Desmet. A role-based implementation of context-dependent communications using split objects. In *Proceedings of the ECOOP 2006 Workshop on Revival of Dynamic Languages*, 2006.

[42] J. Vallejos, P. Ebraert, B. Desmet, T. van Cutsem, S. Mostinckx, and P. Costanza. The Context-Dependent Role Model. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2007)*. Springer LNCS, 2007.

[43] M. Veit and S. Herrmann. Model-View-Controller and ObjectTeams: A perfect match of paradigms. In Akşit [3], pages 140–149.

[44] P. Zumkehr. Changeboxes — modeling change as a first-class entity. Master's thesis, University of Bern, Feb. 2007.

ABOUT THE AUTHORS

Robert Hirschfeld is a Professor of Computer Science at the Hasso-Plattner-Institut (HPI) 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 http://www.swa.hpi.uni-potsdam.de/.

Pascal Costanza has a Ph.D. degree from the University of Bonn, Germany, and works as a research assistant at the Programming Technology Lab (Prog) of the Vrije Universiteit Brussel, Belgium. He can be reached at pascal.costanza@vub.ac.be. See also http://p-cos.net.

Oscar Nierstrasz is a Professor of Computer Science at the Institute of Computer Science (IAM) of the University of Bern. He completed his B.Math at the University of Waterloo in 1979 and his M.Sc. in 1981 and his Ph.D. in 1984 at the University of Toronto. He can be reached at oscar.nierstrasz@acm.org. See also http://www.iam.unibe.ch/∼oscar.

A CONTEXTJ⋆

In this section, we present an implementation of a subset of ContextJ which is realized in plain Java. ContextJ as used throughout this paper would require an extension of the Java programming language, together with a new compiler and possibly an extension of the Java Virtual Machine. As a proof of concept, we focus here on a subset of ContextJ that is necessary to support the example programs presented in this paper. Furthermore, we require a few idioms to be followed in client code that could otherwise be automatically generated in a proper ContextJ compiler or preprocessor. An advantage of this approach is that the implementation presented in this appendix can be run in a standard Java environment without any extensions.[2] To distinguish between ContextJ as presented in the previous sections and ContextJ as implemented in this appendix, we refer to the latter as ContextJ⋆.

Actor-dependent Behavior Variations in ContextJ⋆

We introduce ContextJ⋆ by first showing a reimplementation of the example given in Section 5 for actor-dependent behavior variations, depicted in Figure 9.

To support layered definitions in Java classes, a program has to create instances of the ContextJ⋆ class `Layer` to represent layers as first-class objects (lines 5-8). Each class that wants to support context-dependent bahavioral variations has to provide a context interface for (at least) the methods that should behave differently according to context (lines 12-14) and has to implement that context interface (line 21). Such a class has to define an object-specific container for context-dependent layer definitions of the generic type `LayerDefinitions`, parameterized with the previously defined context interface (lines 35-36). The methods that need to exhibit context-dependent behavior have to forward calls to that container using the container-specific method `select` (lines 38-40). This is always possible because `select` is guaranteed to return an object implementing the context interface on which the container is parameterized.

A `LayerDefinitions` container supports two further methods `define` and `next`. With `define`, a context-dependent implementation for a given layer can be defined for the class at hand: It takes a layer instance – for example the ContextJ⋆-defined `RootLayer` or any of the application-defined layers (here from lines 5-8) – and an instance of another class implementing the respective context interface, typically in the form of an anonymous class instance (for example lines 42-48). Such definitions are conveniently provided in an initializer that runs as part of any constructor (lines 42-65). The `next` method of a `LayerDefinitions` container can be used to call the next most specific method in a current set of active layers. In other words, the idiom `layers.next(this).someMethod(...)` picks the next most specific layer definition and calls the indicated method, similar to an invocation of `proceed` in

[2]The code presented in Appendix A has been tested with JDK 1.5.0.

```java
// File: Layers.java

import static be.ac.vub.prog.contextj.ContextJ.*;

public class Layers {
  public static final Layer Address = new Layer();
  public static final Layer Employment = new Layer();
}

// File: IPerson.java

public interface IPerson {
  public String toString();
}


// File: Person.java

import static be.ac.vub.prog.contextj.ContextJ.*;

public class Person implements IPerson {

  private String name;
  private String address;
  private IEmployer employer;

  public Person(String newName,
                String newAddress,
                IEmployer newEmployer) {
    this.name = newName;
    this.address = newAddress;
    this.employer = newEmployer;
  }

  private LayerDefinitions<IPerson> layers =
    new LayerDefinitions<IPerson>();

  public String toString() {
    return layers.select().toString();
  }

  { layers.define(RootLayer,
      new IPerson() {
        public String toString() {
          return "Name: " + name;
        }
      }
    );

    layers.define(Layers.Address,
      new IPerson() {
        public String toString() {
          return layers.next(this) + "; Address: " + address;
        }
      }
    );

    layers.define(Layers.Employment,
      new IPerson() {
        public String toString() {
          return layers.next(this) + "; [Employer] " + employer;
        }
      }
    );
  }
}
```

```java
// File: IEmployer.java

public interface IEmployer {
  public String toString();
}


// File: Employer.java

import static be.ac.vub.prog.contextj.ContextJ.*;

public class Employer implements IEmployer {

  private String name;
  private String address;


  public Employer(String newName,
                  String newAddress) {
    this.name = newName;
    this.address = newAddress;
  }


  private LayerDefinitions<IEmployer> layers =
    new LayerDefinitions<IEmployer>();

  public String toString() {
    return layers.select().toString();
  }

  { layers.define(RootLayer,
      new IEmployer() {
        public String toString() {
          return "Name: " + name;
        }
      }
    );

    layers.define(Layers.Address,
      new IEmployer() {
        public String toString() {
          return layers.next(this) + "; Address: " + address;
        }
      }
    );
 }
}
```

Figure 9: An implementation of the example from Figure 4 in ContextJ⋆.

```
import static be.ac.vub.prog.contextj.ContextJ.*;

public class Test {
  public static void main(String[] args) {
    final Employer vub      = new Employer("VUB", "1050 Brussel");
    final Person somePerson = new Person("Pascal Costanza", "1000 Brussel", vub);

    with(Layers.Address, Layers.Employment).eval(
      new Block() {
        public void eval() {
          System.out.println(somePerson);
        }
      }
    );
  }
}

Output:  Name: Pascal Costanza; Address: 1000 Brussel;
         [Employer] Name: VUB; Address: 1050 Brussel
```

Figure 10: Direct layer activation in ContextJ⋆.

ContextJ as presented earlier. Here, the method to be invoked will be the implicit call to `toString()` (lines 53 and 61).

The idiomatic code that could be generated by a preprocessor consists of:

- the declaration of an instance variable that holds the container for layer definitions (lines 35-36),
- the forwarding calls to that container based on the context interface (lines 38-40),
- the calls to `define` in an initializer based on layer declarations (lines 42-65),
- and the idiom `layers.next(this)` based on calls to `proceed` (lines 53 and 61).

A code fragment showing the activation of the layers introduced in the example above is shown in Figure 10 which produces the same result as the original code in Figure 5. ContextJ⋆ provides the (global static) method `with` that takes any number of `Layer` instances and returns an object that allows execution of a block of code in a scope in which these layers are active, through the method `eval`.

An Implementation of ContextJ⋆

An implementation of the ContextJ⋆ constructs used here is shown in Figure 11. It consists of a class `ContextJ` that provides a number of `static` members that can be conveniently used in any program via a static import. A layer is just an instance of an empty class `Layer` (line 7), for example the `RootLayer` (line 9). The set of currently

```java
package be.ac.vub.prog.contextj;

import java.util.*;

public class ContextJ {

  public static class Layer {};

  public static final Layer RootLayer = new Layer();

  private static ThreadLocal<LinkedList<Layer>> activeLayers = new ThreadLocal<LinkedList<Layer>>() {
    protected LinkedList<Layer> initialValue() {
      LinkedList<Layer> ll = new LinkedList<Layer>();
      ll.addFirst(RootLayer);
      return ll;
    }
  };

  public static interface Block {
    public void eval();
  }

  public static class WithEvaluator {
    private Layer[] layers;

    private WithEvaluator(Layer[] layers) {this.layers = layers;}

    public void eval(Block block) {
      LinkedList<Layer> currentActiveLayers = activeLayers.get();
      LinkedList<Layer> clonedActiveLayers = new LinkedList<Layer>(currentActiveLayers);
      for (Layer layer: layers) {
        currentActiveLayers.remove(layer);
        currentActiveLayers.addFirst(layer);
      }
      try {block.eval();} finally {activeLayers.set(clonedActiveLayers);}
    }
  }

  public static WithEvaluator with(Layer... layers) {
    return new WithEvaluator(layers);
  }

  public static class LayerDefinitions<Definition> {

    private IdentityHashMap<Layer,Definition> layerDefinitionMap = new IdentityHashMap<Layer,Definition>();
    private IdentityHashMap<Definition,Layer> definitionLayerMap = new IdentityHashMap<Definition,Layer>();

    public void define(Layer layer, Definition definition) {
      layerDefinitionMap.put(layer, definition);
      definitionLayerMap.put(definition, layer);
    }

    public Definition select() {
      for (Layer l: activeLayers.get()) {
        Definition definition = layerDefinitionMap.get(l);
        if (definition != null) return definition;
      }
      return null;
    }

    public Definition next(Layer layer) {
      LinkedList<Layer> currentActiveLayers = activeLayers.get();
      int index = currentActiveLayers.indexOf(layer);
      List<Layer> nextActiveLayers = currentActiveLayers.subList(index+1, currentActiveLayers.size());
      for (Layer l: nextActiveLayers) {
        Definition definition = layerDefinitionMap.get(l);
        if (definition != null) return definition;
      }
      return null;
    }

    public Definition next(Definition definition) {
      return next(definitionLayerMap.get(definition));
    }
  }
}
```

Figure 11: An implementation of ContextJ⋆ in plain Java.

active layers is stored in a ThreadLocal private variable activeLayers that contains only the RootLayer by default (lines 11-17). This variable is ThreadLocal to ensure that different threads can have different sets of active layers without interfering with each other.

An auxiliary interface Block is introduced to support execution of code blocks for some context (lines 19-21).[3] An auxiliary class WithEvaluator takes care of handling layer activation and deactivation around the execution of a code block (lines 23-37). It stores an array of layers to be activated in an instance variable layers (lines 24 and 26).

The method eval clones the set of currently active layers (line 30), and subsequently activates each layer in the stored array of layers (lines 31-34). To this end, each such layer is first removed from the set of currently active layers (line 32) and then added to the front (line 33). The prior removal of a layer ensures that a layer is never activated more than once, and the addition to the front ensures that the order of active layers always respects the order of activation during program execution.

After layer activation, the code block passed as an argument to eval is executed, and finally the set of active layers is restored to the previously cloned set (line 35).

The static method with takes any number of layers and returns an instance of WithEvaluator initialized with these layers.[4]

Finally, ContextJ$\star$ provides a container type LayerDefinitions to be parameterized with a context interface declaring the context-dependent methods (lines 43-75). This container maps layers to definitions in both directions, from layers to definitions and from definitions to layers (lines 45-46). The method define adds a layer/definition mapping to these two maps (lines 48-51). The method select iterates over the set of currently active layers (line 54), searches for the first layer that maps to a definition and returns that definition (lines 55-56). If no mapping is found, select returns null.

There are two versions of the method next (lines 61-70 and 72-74). One takes an instance of the class Layer and searches for another layer that follows that instance in the set of currently active layers (lines 63-64) and that maps to a definition (lines 65-68). (The search is implemented analogously to that of the method select.) The other version of next takes an instance of the type parameter Definition, picks the corresponding layer from the second map and calls the previous version of next with that layer (lines 72-74). The latter version of next is the one used in the idiom layers.next(this) used for calling the next most specific method in the set of currently active layers.

[3]This is similar to the standard Runnable interface as used in Java's multithreading support.

[4]A similar pair of definitions for layer deactivation WithoutEvaluator and without can be implemented anologously, but is left out here to save space.

Please note the following:

- The return of `null` if no layer is found in the `select` and `next` methods appears to be dangerous at first. However, `RootLayer` is always present by default in the set of current active layers, should never be removed in a `without` construct and should always be mapped to a definition in each context-dependent class. All these requirements can be checked and ensured statically. In other words, it should be straightforward to ensure that there is always a definition.[5]
- The search for definitions is not very efficient. However, we have already shown how caches speed up method lookup in ContextL such that the overhead becomes unnoticeable for non-trivial benchmarks [16]. See the related works section of that paper for hints towards further optimization opportunities.

Conditional Layer Activation in ContextJ⋆

As a final illustration of using ContextJ⋆, we present the examples for conditional layer activation from Figure 6 using ContextJ⋆ in Figure 12. Code section 1/2 repeats the approach in which the code on which layers may need to be activated or not is guarded by an `if` statement. Code section 3 shows how the guard can be inlined as a conditional expression that produces the correct set of layers to be activated depending on context and passes that set of layers to `with`. Here we take advantage of the fact that, as in ContextS, layers are first-class entities. The general idiom is presented in code section 4 in Figure 12.

```
// -[1/2]- Direct layer activation with code replication

if (currentUser().verboseMode())
  with(Layers.Adress).eval(new Block() {
    public void eval() { for (Person p: personList) System.out.println(p); }
  });
else for (Person p: personList) System.out.println(p);

// -[3]- Direct layer activation without code replication

with(currentUser().verboseMode() ?
     new Layer[] {Layers.Adress} : new Layer[] {}).eval(new Block() {
  public void eval() { for (Person p: personList) System.out.println(p); }
});

// -[4]- Direct layer activation based on context object

with(context.computeLayers()).eval(new Block() {
  public void eval() { for (Person p: PersonList) System.out.println(p); }
});
```

Figure 12: Conditional layer activation examples from Figure 6 in ContextJ⋆.

[5]A proof of soundness is future work, though.