In Proceedings of the JSSST Annual Conference 2009, 2D-1,
Shimane University, Matsue, Shimane, Japan, September 16-18, 2009.

ContextJ: Context-oriented Programming with Java

Malte Appeltauer Robert Hirschfeld Michael Haupt Hidehiko Masuhara

Context-oriented programming (COP) allows for the modularization of context-dependent behavioral variations. So far, COP has been implemented for dynamically-typed languages such as Lisp, Smalltalk, Python, and Ruby relying on reflection mechanisms, and for the statically-typed programming language Java based on a library and a pre-processor. ContextJ is our COP implementation for Java. It properly integrates COP's layer concept into the Java type system. ContextJ is compiler-based. As confirmed by a benchmark and a case study, it provides both better performance and higher-level abstraction mechanisms than its Java-based predecessors. In this paper, we present the ContextJ language and explain its constructs and semantics. Further, we give an overview of the implementation of our compiler and discuss run-time benchmarks.

1 Introduction

The separation of *crosscutting concerns* (CCCs) is an important issue of modern program language design. A CCC is program behavior that cannot be adequately modularized with respect to the other parts of a system [26]. Such concerns are typically scattered over an application's modules and hinder software evolution and maintenance. Some programming paradigms, e. g., *aspect-oriented programming* [20] (AOP), *feature-oriented programming* [7] (FOP), *adaptive object-oriented programming* [23], and *context-oriented programming* [18] (COP) offer language abstractions to cope with CCCs.

COP is a novel approach to the dynamic composition of CCCs, making it easier for example to adapt a user interface based on the current user's profile or to instrument a server-side application to record events for settlement according to a customer's current rate plan. COP introduces *layers*, an encapsulation mechanism for behavioral variations that can crosscut several modules of an application. Behavioral variations are represented by *partial method definitions* that can dynamically override or extend their respective base methods. Partial methods are grouped into layers. Layers can be dynamically composed with other layers, allowing fine-grained control over an application's run-time behavior. A broad introduction to COP is provided in the other literature [18]. The approach has been implemented mainly for dynamic languages, such as Lisp [9], Smalltalk [17], Python [31], and Ruby [29].

The implementations of COP extensions to dynamic languages extensively use the languages' meta-level capabilities. Meta-programming is a powerful means; for instance, it allows for manipulating method dispatch at run-time, which is necessary for dynamic method overriding, a key feature of COP. However, the expressiveness of

Malte Appeltauer, Robert Hirschfeld, and Michael Haupt, Hasso Plattner Institute, University of Potsdam, Germany, ,

増原英彦, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, University of Tokyo, Japan,

meta-programming does not come for free. As discussed in [1], the meta-level implementations of COP introduce a significant performance decrease to method lookup and execution of their host languages.

Based on our experiences with previous COP implementations, we stated the following two requirements for a Java language extension. First, run-time performance must be considered. Since a reflection-based implementation is not an option, we need to use an alternative implementation strategy. Second, the language extension should be fully integrated into the Java language. This includes a intuitive syntax and integration of the COP concepts into the Java type system.

In this paper, we present a ContextJ language specification and a compiler-based implementation that fulfills our requirements. We introduce ContextJ's language constructs and give an overview of the compiler design and implementation. Furthermore, we discuss run-time benchmark evaluations and a case study.

Our paper is structured as follows. Section 2 gives an overview of context-oriented programming. Section 3 explains its features for the modularization and run-time composition of crosscutting concerns. The ContextJ compiler is presented in Section 4, along with some micro-benchmarks. We discuss related work in Section 5 and finally summarize the paper in Section 6.

2 Context-oriented Programming

2.1 Overview

The COP paradigm features a new approach to software modularization by supporting an explicit representation of context-dependent functionality that can be dynamically activated or deactivated.

Below, we introduce basic notions of COP relevant in this paper.

COP assumes *context* to be *everything that is computationally accessible*, such as a variable's value, control flow properties, or even external events. Based on these primitives, context can be modeled for more complex information such as personalization, security settings, or location-awareness.

Layers are a modularization concept orthogonal to classes, in which crosscutting context-specific functionality can be encapsulated. Layers can range over several classes and contain *partial method definitions* that implement behavioral variations. To distinguish between the different kinds of method definitions, we introduce the terms *plain method definition* and *layered method definition*. A plain method denotes a method whose execution is not affected by layers. Layered methods consist of a *base method definition*, which is executed when no active layer provides a corresponding partial method, and at least one partial method definition.

Layers are *composed* at run-time. Their partial method definitions can be executed before, after, around, or instead the base method definition. More than one layer of a composition may provide a partial definition of the same method, therefore, a partial method can *proceed* to the next partial definition in the composition or, if no adequate variation exists, to the base method definition.

Layer composition is controlled *per thread* and is by default scoped to the dynamic extent of a block of statements. This fine-grained dynamic composition is essential for the development of context-dependent systems.

2.2 Java-based COP

The first ideas about a ContextJ language have been presented in [10] to improve the accessibility of the *ContextL/Lisp* code discussed in that paper. The authors introduced ContextJ syntax only in a pseudo-code manner and neither provided a

```
1 public class Account {
2   private int accountNumber;
3   private float balance;
4   public Account(int accNr) {
5     accountNumber = accNr;
6   }
7   public void credit(float amount) {
8     balance = balance + amount;
9   }
10  public void debit(float amount) {
11    balance = balance - amount;
12  }
13  public float getBalance() {
14    return balance;
15  }
16 }
17 public class TransferSystem {
18   public void transfer(Account from, Account to,
19                        float amount) {
20     from.debit(amount);
21     to.credit(amount);
22   }
23 }
```

Listing 1 Bank accounts and transfers.

feature-complete syntax nor a language specification.

That ContextJ draft has not been implemented, however there exists a proof-of-concept implementation called *ContextJ** [18]. It is a Java 5 library that implements the core concepts of COP, i. e., layer definition and activation. This implementation demonstrates that COP can be supported by means of Java without any extension to the syntax or semantics of the language. However, the proper use of ContextJ* requires developers to follow several idioms within their code, leading to complex implementations in ContextJ* programs.

ContextLogicAJ [4] [2] is an aspect-oriented precompiler that offers a more convenient syntax than ContextJ*. It is based on an aspect library implemented in *LogicAJ* [22], a generic aspect language. With ContextLogicAJ, we experimented with alternative implementation and transformation techniques to transform COP semantics into an object-oriented language.

Both implementations remain prototypes and only provide limited COP support. Furthermore, they do not extend Java's syntax with dedicated constructs but expect the programmer to also follow several idioms, which makes code fragile and with that not applicable to production scenarios.

3 ContextJ Language

In the following, we introduce ContextJ language constructs and successively extend the example presented in Listing 1 with context-specific functionality. The class `Account` contains methods to credit or debit money. A `TransferSystem` handles the transfer of an amount of money from one account to another.

Throughout this section, we show the implementation of two context-dependent concerns, encryption and logging, by means of ContextJ. We assume that these concerns should not be statically applied to the system, but rather dynamically composed whenever necessary. The syntax production rules are specified in Extended Backus-Naur Form (EBNF), where terminals are shown in `fixed font`. ContextJ extends the set of Java terminal symbols with **`layer`**, **`with`**, **`without`**, **`proceed`**, **`before`**, and **`after`**. We omit standard Java elements by using "..." and present only the ContextJ constructs and their entry points into the Java syntax [14].

3.1 Modularization

Layer. ContextJ implements the *layer-in-class* style [18]; that is, layers are defined within classes, and classes thereby carry their own context-specific variations. The syntactic structure of the construct is shown below.

```
1  import layer EncryptionLayer;
2  import layer LoggingLayer;
3
4  public class Account {
5    ...
6    layer EncryptionLayer {
7      public void credit(int am) {
8        proceed(RSA.decrypt(am));
9      }
10     public void debit(int am) {
11       proceed(RSA.decrypt(am));
12     }
13     public int getBalance() {
14       return RSA.encrypt(proceed());
15     }
16   }
17   layer LoggingLayer {
18     after public void credit(int am) {
19       Logger.logCredit(this, am);
20     }
21     after public void debit(int am) {
22       Logger.logDebit(this, am);
23     }
24     public int getBalance() {
25       int balance = proceed();
26       Logger.logBalanceRequest(this, balance);
27       return balance;
28     }
29   }
30 }
31 ------------------------------------------
32 import layer EncryptionLayer;
33 import layer LoggingLayer;
34
35 public class TransferSystem {
36   ...
37   layer EncryptionLayer {
38     public void transfer(Account from, Account to,
39                          int amount) {
40       without (EncryptionLayer) {
41         proceed(from, to, RSA.encrypt(amount));
42       }
43     }
44   }
45   layer LoggingLayer {
46     after public void transfer(Account from,
47                         Account to, int amount) {
48       Logger.logTransfer(from, to, amount);
49     }
50   }
51 }
```

Listing 2 Layers for encryption and logging.

ClassBodyDeclaration ::=
  ... | *LayerDefinition*
LayerDefinition ::=
  `layer` *Identifier* { *PartialMethodDefinition** }

A layer consists of an identifier and a list of *partial method definitions*. A partial method definition's signature must correspond to that of a method of the enclosing class or its superclass. Final methods cannot be extended by layers.

Layer Identifier. Layers are referenced by layer identifiers that must be made visible to the compilation unit by using a *layer import declaration*, corresponding to type import declarations.

ImportDeclaration ::=
  ... | *LayerImportDeclaration*
LayerImportDeclaration ::=
  `import layer` *Identifier* ;

Partial Method Definitions. Layer definitions can contain *partial method definitions*. A partial method definition of a method M overrides the default definition of M during the activation of its layer. Partial method definitions allow different strategies to proceed to their corresponding method. Besides the default around behavior, partial methods can provide functionality that should be executed *before* or *after* a particular method. This intention can be expressed with the modifiers `before` and `after` for partial methods, denoting that their behavior should be executed before or after the method execution. An `after` method is always executed after the original method, even if it throws an exception. This semantics corresponds to *after returning or throwing* advice of AspectJ-like languages.

PartialMethodDefinition ::=
  [`before` | `after`] *MethodDeclaration*

For explicit invocation of the next partial method definition (or the default method), the built-in pseudo method `proceed` can be used. Both the return type and the expected arguments of `proceed` conform to the method's signature.

Expression ::=
  ... | *Proceed*
Proceed ::=
  `proceed(` *ArgumentList* `)`

Listing 2 depicts the separate declaration of two layers that implement crosscutting concerns. For example, the definition of `EncryptionLayer` in `Account` (Lines 5–15) contains partial definitions of methods that encrypt or decrypt method parameters and then call the next partial definition with the encrypted values. The same layer provides a partial definition of a method within `TransferSystem` (Lines 35–51).

The partial methods in Lines 7–15 and 24–28 invoke the next definition by calling `proceed` with the new parameters. `LoggingLayer` (Lines 17–29, 45–50) introduces logging functionality to the methods. Some of its partial method definitions contain the `after` modifier, which means that they are executed after the computation of their next partial definition. To use layer identifiers in a class, the enclosing compilation unit must declare them first (Lines 1–2 and 32–33).

3. 2 Dynamic Layer Composition

Layer Activation. To control scoped layer activation, ContextJ introduces a new block statement, `with`, that can be used in method bodies. The `with` block provides an argument list that contains the identifiers of the layers to be activated. These layers are only active for the *dynamic extent* of the `with` block. This implies that the activation of a particular layer is confined to the threads in which the layer was explicitly activated. Layer activation does not propagate to new threads; they start with no layers being active.

```
1  public void transfer100(Account from,
2                          Account to) {
3    ...
4    with(LoggingLayer) {
5      with(EncryptionLayer) {
6        transferSystem.transfer(from, to, 100);
7      }
8    }
9  }
10 ------------------------------------------
11 public void transfer100(Account from,
12                         Account to) {
13   ...
14   with(LoggingLayer, EncryptionLayer) {
15     transferSystem.transfer(from, to, 100);
16   }
17 }
```

Listing 3 Nested layer activation.

Block ::=
  ... | *LayerActivation*
LayerActivation ::=
  `with(`*ArgumentList*`)` {*BlockStatement**}

Like standard Java block statements, `with` statements can be nested. The list of active layers is then extended with the arguments of the inner layer activation. If more than one active layer provides a partial definition for a method, the order of layer activation defines the proceed chain. The list of active layers is traversed according to the *last-in-first-out* principle: the most recently activated layer is visited first. When a layer is activated or deactivated more than once, only its most recent activation or deactivation is effective.

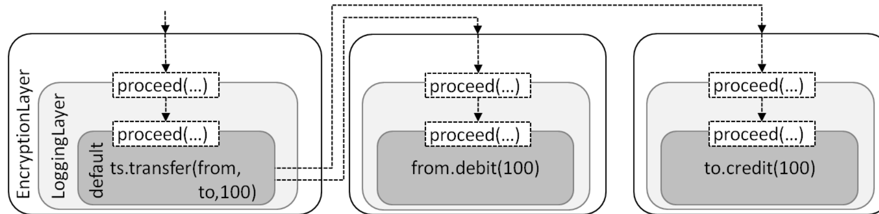

図 1 **Progression of a method invocation through a list of active layers.**

ContextJ supports the *direct* and *indirect* enumeration of a sequence of layers to be activated. Layer identifiers can be directly passed to the argument list.

Listing 3 shows two alternative notations for a direct layer activation. Lines 1–8 show two nested `with` blocks that consecutively activate `LoggingLayer` and `EncryptionLayer`. Lines 10–15 contain the same activation using a list of layer identifiers in a single `with` block.

Figure 1 illustrates the execution of `transfer` in Lines 5 and 13 in Listing 3. The invocation is first dispatched to `EncryptionLayer`, then to `LoggingLayer`, and finally to the base method. The base method of `transfer` invokes credit and debit methods on its `Account` parameters. Both active layers also provide partial methods for them, thus the method calls again pass the layers, as depicted in Figure 1.

Often, the computation of the layers to be used is either complex, or the layers cannot be directly specified at compile-time. For more flexibility, expressions of type `Layer`, `Iterable<Layer>`, or `Layer[]` can be used as arguments. If the `with` arguments are evaluated to an empty list (or `null`), no layer will be activated. Listing 4 exemplifies such an indirect activation by a method call used as `with` argument.

```
1  public void debit100(Account account) {
2    with(debitCompositon(account)) {
3      account.debit(100);
4    }
5  }
6
7  public List<Layer> debitCompositon(Account acct) {
8    List<Layer> layers = new ArrayList<Layer>();
9    if(securityLevelHigh(acct))
10     layers.add(EncryptionLayer);
11   if(transactionLoggingAllowed(acct))
12     layers.add(LoggingLayer);
13   return layers;
14 }
```

Listing 4 First-class layers.

Layer Deactivation. We provide a means to express the exclusion of a certain layer from a composition. This is because, if several layers provide a partial definition of a certain method, it may be possible that these definitions interfere with each other. The `without` block construct works contrariwise to `with` in the sense that layers specified by `without` are deactivated for its dynamic extent. All other properties regarding thread locality and nesting hold as described for layer activation above.

Block ::=
    ... | *LayerDeactivation*
LayerDeactivation ::=
    `without(`*ArgumentList*`)` {*BlockStatement**}

Listing 5 contains a modified version of the partial method of `TransferSystem.transfer` that uses

```
1 layer EncryptionLayer {
2   public void transfer(Account from,
3     Account to, int amount
4   ) {
5     without(LoggingLayer) {
6       proceed(from, to, RSA.encrypt(amount));
7     }
8   }
9 }
```

Listing 5 Layer deactivation.

`without` to prevent the logging layer from monitoring the transaction.

3.3 Reflection API

With the constructs presented so far we are able to handle most common scenarios for behavioral variations. For situations requiring special reasoning about layer, we provide a reflection API that gives access to inspect and manipulate layers, their composition and their partial methods at run-time. The API consists of three classes of the `contextj.lang` package, namely `Layer`, `Composition`, and `PartialMethod`. The superclass of all layers, `Layer`, provides methods to access a layer's enclosing composition and partial method definitions. `Composition` objects allow access to their layers and the (de-)activation of layers. `PartialMethod` is the metaclass of partial methods, corresponding to Java's `java.lang.reflect.Method` class. As `Method`, it inherits from `AccessibleObject` and implements the `Member` interface, which are both defined in the package `java.lang.reflect`. Table 1 describes the API methods.

As an example for the use of the API, we want to assert that no other layer provides a partial definition for `transfer`. Listing 6 presents an implementation of such behavior. First, we access the current composition (Line 3) and retrieve an array

```
1 layer EncryptionLayer {
2   public void transfer( ... ) {
3     Composition comp = Composition.current();
4     Layer[] ls = comp.getLayers();
5     String signature = /*this methods signature*/;
6     for(Layer l : ls ) {
7       if ( (l != EncrpytionLayer) &&
8            (l.providesPartialMethodFor(signature)))
9         throw new RuntimeException(
10          "Method overriding by layer forbidden.");
11    }
12    /* do the encryption */
13  }
```

Listing 6 Use of reflection API.

of all active layers (Line 4). For each active layer except `EncryptionLayer` we check if it provides a partial definition of `transfer` (Lines 6-7). If it does, we throw a runtime exception.

4 Implementation and Evaluation

We developed a compiler for ContextJ because the reflection-based implementation approaches (see Section 5) taken for COP extensions to dynamic languages are not suitable for Java.

4.1 Layer-aware Message Dispatch

Since we want to use ContextJ with existing Java tools and environments, our compiler is byte code compatible with Java. To generate plain Java byte code from ContextJ source code, we developed a translator from ContextJ's abstract syntax tree (AST) to that of Java. This translator, as described in the following, is implemented as re-write rules that are executed during compilation.

First, we describe the general steps of layer-aware method lookup at runtime. For a call to a method M and a list of active layers L:

1. Find the last layer $L_i \in L$ that contains a partial method definition (M_{L_i}) for method M.
2. If a M_{L_i} exists, execute it.
3. If M_{L_i} contains a `proceed` expression, lookup

contextj.lang.Layer	
`static Layer forName(String)`	Returns the layer associated with the given string name
`Composition getComposition()`	Returns the enclosing layer composition
`boolean isActive()`	Returns `true` if the layer is activated
`boolean providesPartialMethodFor(String)`	Determines if the layer provides a partial definition for a method with signature represented by the parameter
`PartialMethod[] getPartialMethods()`	Returns an array of `PartialMethod` objects reflecting all the partial methods provided by the layer
`PartialMethod getPartialMethod(String)`	Returns a `PartialMethod` object representing a partial method of the layer with the signature specified by the parameter
contextj.lang.Composition	
`Layer[] getLayers()`	Returns an array of the layers of the composition
`void activateLayer(Layer)`	Activates a layer in the current composition
`void deactivateLayer(Layer)`	Deactivates a layer in the current composition
contextj.lang.PartialMethod	
`Layer getDefiningLayer()`	Returns the layer defining this partial method
`Class getDeclaringClass()`	Returns the declaring class of the partial method
`Class[] getExceptionTypes()`	Returns an array of the exception types
`String getName()`	Returns a string representation of that method
`Class getReturnType()`	Returns the return type of the method
`int getModifiers()`	Returns the Java language modifiers for the method represented by this `Method` object, as an integer
`Object invoke(Object target, Object... args)`	Invokes the underlying partial method on the specified object with the specified parameters

表 1 The ContextJ reflection API.

the next layer $L_x \in L, x < i$ that contains M_{L_x} and repeat Step 2, else continue with Step 4.

4. Execute the original method definition.

The dynamic structure of L can be implemented as an ordered list consisting of layer objects. For the implementation of layer lookup we use inheritance: Each layer L_i is subtype of *ConcreteLayer*, which in turn inherits from *Layer*. If no layer is activated, the layer list only consists of one *Layer* element. For each layered method M, *Layer* provides a delegation method that simply calls M, corresponding to Step 4.

To traverse the layer list in Steps 1 and 3, *ConcreteLayer* overrides these methods and implements a delegation to the next layer in the list. Each L_i that provides a M_{L_i} overrides the delegation method of *ConcreteLayer* with a call to M_{L_i}, which is implemented in the same class as M. Its signature corresponds to M's, except for the first parameter, whose type is L_i. The first parameter allows to distinguish multiple partial definitions of M. Layer activation can be implemented in a simple way: Basically, the `with` block is replaced by two static methods of *Layer* that allow to add and remove items from the list.

Mappings for `Account` and `EncryptionLayer` are shown in Figure 2. Note that the Java source code presented here is not generated but directly transformed into byte code during compilation.

4.2 Compiler Implementation

Syntax Specification. The implementation of the ContextJ compiler is an extension of *JastAddJ* [12], an open Java compiler based on the *JastAdd* [16] compiler framework. Typically, compiler extensions require adaptations in several modules, such as the scanner, parser, abstract syntax tree (AST), and semantic analysis. JastAdd is a modular compiler framework that uses aspect-oriented techniques to encapsulate specifications into dedi-

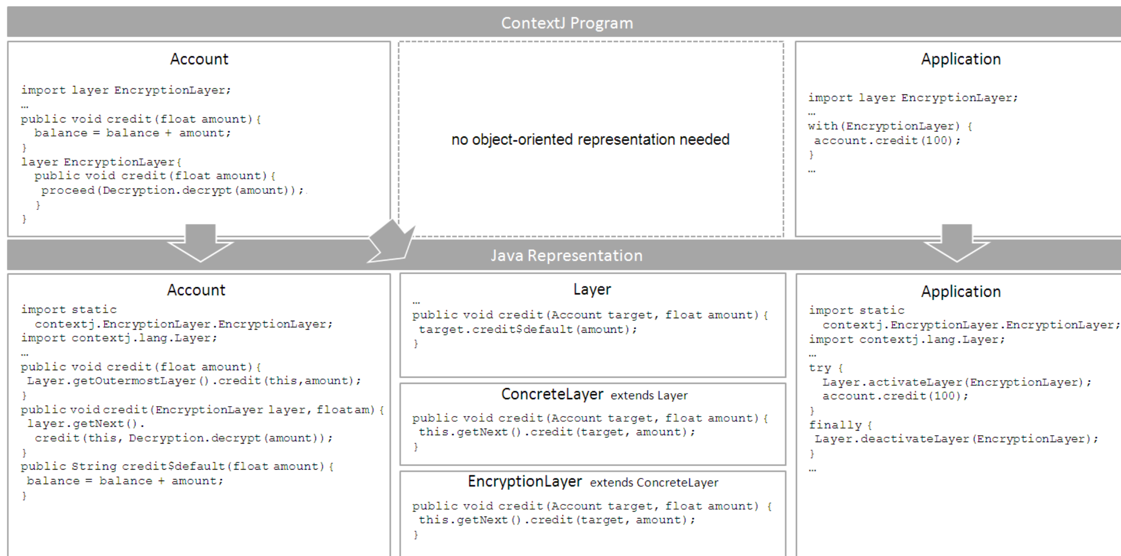

図 2 Mapping of a ContextJ program to Java.

cated modules. During the compiler build process, the separate specifications are woven into one executable compiler.

For lexical analysis, JastAdd employs *JFlex* [21], a scanner generator for Java. Each keyword specification provides a corresponding terminal symbol that can be used in the parser and is woven into the scanner at build-time. This is how the ContextJ keywords are introduced.

JastAdd provides an object-oriented abstract grammar from which the Java AST representation is generated. The abstract grammar does not contain any behavior specification; this is done by separate attribute and equation specifications. For a modularized specification, inter-type declarations are used to extend existing trees. We extend the Java AST definition by node types for layers, partial method definitions, the `proceed` expression, and layer activation and deactivation.

By default, JastAdd uses the Java-based parser generator *Beaver* [11], a LALR(1) parser generator. The system is able to consume the tokens that are generated by JFlex. Beaver accepts a context free grammar, expressed in EBNF, and converts it to a Java class that implements a parser for the language described by the grammar.

AST Transformations. For the implementation of the behavior shown in Section 4. 1, we make use of JastAdd's re-writing facilities. Typically, re-write rules change or replace a certain AST node or subtree with another. We use this technique to translate ContextJ-specific nodes into Java nodes that implement their behavior. For the implementation of layer-aware message dispatch the re-write rules introduce a class for each layer L and several methods for each of L's partial methods.

In the following, we describe the transformation steps to generate these methods.

- For each layer L, a class L^{class} will be created as a subtype of `contextj.lang.Layer`
- A new parameter of type L^{class} is inserted into the parameter list of each partial method definition M^L. Subsequently, M^L is moved to the enclosing class. When all partial methods of L have been transformed, L is removed from the member list of its enclosing class.

- For each M^L a forwarding method $M^{forward}$ is created in L^{class}. It calls M^L with its own instance as first parameter.
- The body of a base method M^{base} is moved to a new method M_base.
- For each M^L a default forwarding method $M^{forward}$ is created in `contextj.lang.Layer`. It calls $M^{forward}$ on the next layer of the composition. If the composition does not contain any more layers it calls M_base with its own instance as first parameter.
- The body M^{base} will be replaced by a call to $L^{class}_{first}.M^{forward}$, where L^{class}_{first} is the outermost layer in the thread local composition.

In addition to this transformations, the compiler provides auxiliary transformations for static, private, or protected methods. Figure 2 gives an example of ContextJ syntax and its transformation into Java.

Finally, the compiler generates byte code for the transformed layers. The application can then be executed as a plain Java program.

4.3 Benchmarks

This section discusses our run-time measurements, based on the *Java Grande Forum Benchmark Suite* [8], for which we developed, in the fashion of [15], a set of micro-benchmarks to assess the performance of layer-aware method dispatch. The micro-benchmarks were run on an 1.8 GHz dual core Intel Core2Duo with 2 GB main memory running on Windows XP. All benchmarks are executed once before the actual measurement to assure that virtual machine optimizations are performed.

4.3.1 Plain vs. Layered Methods

In order to measure the overhead of the execution of a layered method compared to an identical plain method, we set up a micro-benchmark that executes different types of plain methods and layered methods without active layers. The benchmark includes synchronized and non synchronized instance and class methods. Besides self-invocations, the benchmark considers calls to other classes. All methods increment a class variable.

Figure 4 illustrates the results of this benchmark. In all tests, plain methods are significantly faster than their layered counterparts; the latter are two to five times slower. The overhead is caused by layer lookup, which is executed even if no layer is active. The lookup requires at least three additional method invocations plus access to the thread-local layer list.

4.3.2 Layer-aware Message Dispatch

A further set of benchmarks measures the overhead caused by the execution of an increasing number of partial methods. We again compare the throughput of plain methods and layered methods. The measurement consists of 15 plain methods (`m01-m15`) and integer fields (`c01-c15`), where each method increments one more field than its predecessor, so that method `m01` increments `c01` and `m15` increments the fields `c01-c15`. The benchmark version using layers contains one base method `m` that increments `c01`, and 14 layers. Each layer provides a partial definition for `m` that increments one distinct field and then proceeds to the next layer.

The results are presented in Figure 5. The number of invocations per millisecond of the layered method decreases with an increasing number of layers from approximately 40,000 to 10,000, which is a performance decrease of 75 %. The plain method calls per millisecond range from 300,000 for a method that increments one field down to 77,000 for a method incrementing 15 fields. Again, the performance decrease equals approximately 75 %.

The plain methods are highly optimized by the Java virtual machine in the first four runs of our benchmark, where up to four fields are incremented. The succeeding runs of five and more field increments are approximately three times slower,

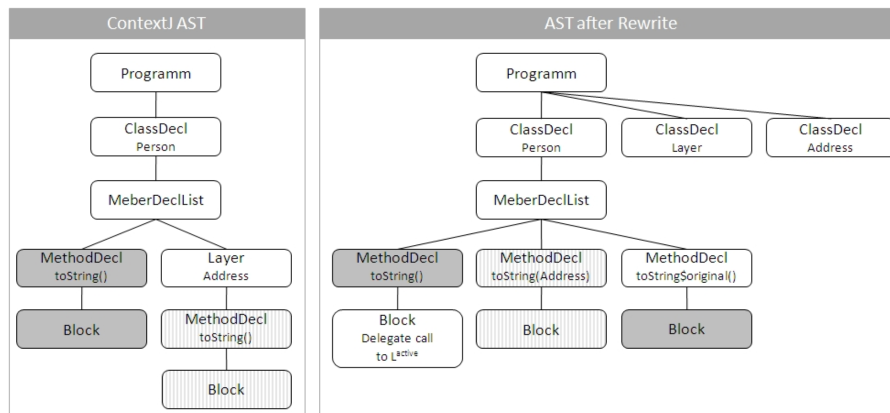

図 3 AST transformation of ContextJ nodes to Java nodes.

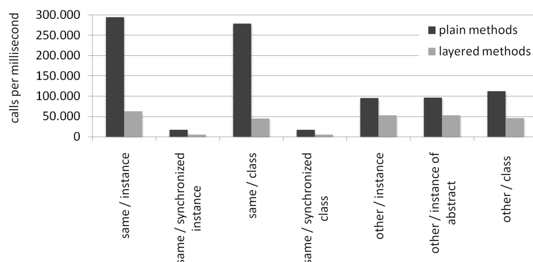

図 4 Different types plain and layered methods.

indicating that the virtual machine stops some optimizations of blocks containing more than four field increments.

The layered methods are executed significantly slower. The Java code of layered message lookup generated by the ContextJ compiler contains thread-local method invocations that cannot be easily optimized by the Java VM. However, except for the overall overhead, intensive use of layers increases execution time of layered methods only proportional to plain methods. Nevertheless, future work on ContextJ must consider performance optimizations.

4.4 Other COP Implementations for Java

The preceding measurements compare the runtime behavior of ContextJ with Java. Since one goal of our compiler-based implementation is to provide an adequate *performant* COP implementation in Java, we applied the previous benchmark setting to ContextJ and the two preceding implementations, namely ContextJ* and ContextLogicAJ. The results are presented in Figure 6.

ContextJ and ContextLogicAJ exhibit roughly equal performance characteristics. We expected this result since both approaches transform COP syntax into (almost the same) plain Java code at compile-time or weaving time, respectively. ContextJ supports however more features and comes with a dedicated, more declarative syntax than ContextLogicAJ.

ContextJ and ContextLogicAJ perform significantly better than the Java 5 based ContextJ* approach. With more than one active layer, ContextJ* constantly processes approximate 1,500 method calls per millisecond. This is 6 to 16 times slower than ContextJ and ContextLogicAJ.

4.5 Case Study

As a case study, we implemented *CJEdit* [3], a simple *programming environment* for ContextJ in ContextJ. The editor provides syntax highlighting, an outline view, and a compilation/execution toolbar. CJEdit also allows to comment Con-

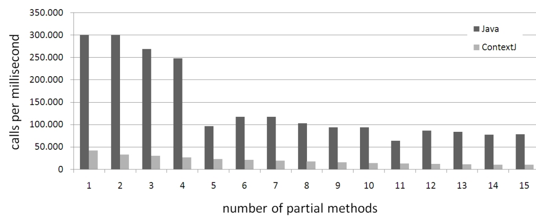

図 5 **Increasing number of active layers.**

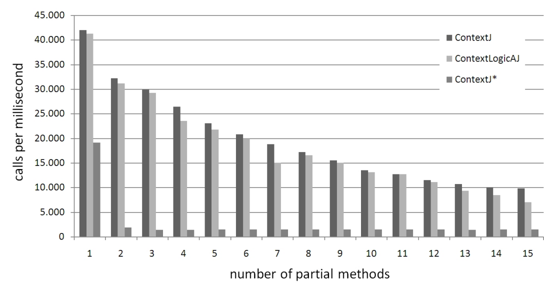

図 6 **Comparison of Java COP extensions.**

textJ compilation units using rich text. For this task, the editor provides *rich text formatting features*. Through the combination of rich text and source code, CJEdit documents are single-source, executable representations of code and documentation.

Both activities require different functionality, so our application supports focusing on the actual task at hand by offering only relevant tools, menus, and widgets. A switch between the text editing and programming features is either directly triggered by the user, or on text cursor change: While writing new text, the user can enter the programming mode by pushing a toolbar button. Whenever the text cursor is moved through the document from text to code and vice versa, the GUI elements are changed accordingly also.

CJEdit is implemented using ContextJ and the *Qt Jambi GUI Framework* [25]. It consists of approximately 1400 lines of code, where most parts are written with plain Java constructs and the help of the Qt GUI Designer. The de-/activation of task-specific user interfaces and behavior are implemented in ContextJ. The system contains distinct layers that encapsulate rich text and programming widgets such as toolbars and their corresponding behavior.

5 Related Work

We discuss related work in two areas; namely existing implementations of context-oriented programming and approaches to the modularization of crosscutting concerns for the Java programming language.

5.1 COP Implementations

ContextL [9] [10] was the first COP extension to a programming language. It is based on Lisp and extends the Common Lisp Object System. Layers can be defined for classes, functions and methods. At run-time, layers can be de-/activated for a certain control flow.

Subsequently, several meta-level libraries for dynamic programming languages were developed, namely *ContextS* [17] for Smalltalk, *ContextR* [29] for Ruby, *ContextJS* for JavaScript, *ContextPy* [30] and *PyContext* [31] for Python, and *ContextG* for Groovy. A minimal subset of ContextJ, *cj*, is implemented for the *delMDSOC* kernel [27] [28].

Another approach to context-orientation is *Ambience* and its underlying *Ambient Object System* [13]. It supports behavior adaptations with partial method definitions and context objects, which correspond to COP layers.

The various context-oriented extensions for several other programming languages are implemented using the respective language's meta-level facilities; none of them utilizes bytecode transformation as ContextJ does. An comparison of their language features is provided in [1].

5.2 Modularization of Crosscutting Con-

cerns in Java

Aspect-oriented language extensions also provide constructs for modularizing CCCs [20] in order to decrease code scattering and tangling. Aspect-orientation adopts a specific view on CCCs: A CCC contains functionality that is executed at different *join points*, well-defined points in a program's control flow. The key abstractions of aspect-oriented languages are *pointcuts*, predicates that describe a set of join points, and *advice*, blocks of functionality that can be bound to pointcuts. In COP, CCCs can be expressed as behavioral variations within layers. Layers can either be dedicated modules like aspects (*class-in-layer*), or defined within the classes themselves (*layer-in-class*). Depending on the host language properties, concrete COP implementation support either one or both layer definition styles. ContextJ supports *layer-in-class* and therefore differs from AspectJ-like Java extensions.

AspectJ [19] is an aspect-oriented language extension to Java. It comes with a join point model that includes method calls, executions and field access. Advice blocks allow to extend a join point with additional behavior which can be woven before, after, or around the join point. Aspects are woven at compile or load time and are globally scoped.

CaesarJ [5] comes with an alternative module concept by unifying classes, aspects, and packages. CaesarJ aspects can be deployed at run-time using different kinds of dynamic scope, much like ContextJ layers. CaesarJ supports virtual classes [24], a concept that enables dynamic class extension, depending on the caller's scope. The ability of virtual classes to extend modules is similar to layers. However, class extension with layers is not bound on the caller's module but differs depending on the current layer composition.

Feature-oriented programming (FOP) [7] addresses the process of step-wise refinement for product-line development. The Java-based AHEAD Tool Suite [6] is an implementation of FOP. As programming language, it supports *Jakarta* which extends Java with constructs such as class refinements for static feature-oriented composition. Layers in Jakarta are distinct files describing static class refinements. The foundations of FOP and COP are similar: Both introduce new or alternative program behavior through features or layers, respectively. However, FOP applies compile-time composition of feature variations in contrast to run-time composition as provided by COP.

For further comparison of AOP, FOP, and COP, we refer to [18].

6 Summary

The modularization of CCCs is a well known issue that is addressed by several programming paradigms and language extensions. For their validation and further development they need to be applied to different language domains to assess their usability and expressiveness. In that regard, Java-like languages are an important domain for the assessment of new language abstractions, due to their popularity and use in a wide range of software systems.

In this paper, we present a compiler-based implementation of ContextJ, a language extension for COP to Java. We describe ContextJ's language features that support the modularization and dynamic composition of CCCs. Further, we show the design and implementation of the ContextJ compiler and discuss benchmarks of layer-aware method lookup. Our language is used for the implementation of a context-aware programming environment and proved to be stable in combination with existing Java-based frameworks.

Acknowledgments

We thank Pascal Costanza for his initial work and fruitful discussions on ContextJ, Marvin Killing for his help on CJEdit, and Michael Perscheid for comments on drafts of this paper.

参 考 文 献

[1] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M.: A Comparison of Context-oriented Programming Languages, *COP '09: International Workshop on Context-Oriented Programming*, New York, NY, USA, ACM Press, 2009, pp. 1–6.

[2] Appeltauer, M. and Hirschfeld, R.: Explicit Language and Infrastructure Support for Context-aware Services, *Beiträge der 38. Jahrestagung der Gesellschaft für Informatik*, Lecture Notes in Informatics, Vol. Informatik 2008 - Beherrschbare Systeme dank Informatik, No. 134, München, Germany, Gesellschaft für Informatik, September 2008, pp. 164–170.

[3] Appeltauer, M., Hirschfeld, R., and Masuhara, H.: Improving the Development of Context-dependent Java Applications with ContextJ, *COP '09: International Workshop on Context-Oriented Programming*, New York, NY, USA, ACM Press, 2009, pp. 1–5.

[4] Appeltauer, M., Hirschfeld, R., and Rho, T.: Dedicated Programming Support for Context-aware Ubiquitous Applications, *UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, Washington, DC, USA, IEEE Computer Society Press, 2008, pp. 38–43.

[5] Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K.: Overview of CaesarJ, *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, Vol. 3880(2006), pp. 135–173.

[6] Batory, D.: Feature-Oriented Programming and the AHEAD Tool Suite, *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, Washington, DC, USA, IEEE Computer Society, 2004, pp. 702–703.

[7] Batory, D., Sarvela, J. N., and Rauschmayer, A.: Scaling Step-Wise Refinement, *IEEE Transactions on Software Engineering*, Vol. 30, No. 6(2003), pp. 355–371.

[8] Bull, M., Smith, L., Westhead, M., Henty, D., and Davey, R.: Benchmarking Java Grande Applications, *Proceedings of the Second International Conference on The Practical Applications of Java*, 2000, pp. 63–73.

[9] Costanza, P. and Hirschfeld, R.: Language Constructs for Context-Oriented Programming: An Overview of ContextL, *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, New York, NY, USA, ACM Press, 2005, pp. 1–10.

[10] Costanza, P., Hirschfeld, R., and De Meuter, W.: Efficient Layer Activation for Switching Context-dependent Behavior, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, Lightfoot, D. E. and Szyperski, C. A.(eds.), Lecture Notes in Computer Science, Vol. 4228, Berlin, Heidelberg, Germany, Springer-Verlag, September 19 2006, pp. 84–103.

[11] Demenchuk, A.: Beaver - a LALR Parser Generator, October v0.9.6.1 released 05/2006. http://beaver.sourceforge.net.

[12] Ekman, T. and Hedin, G.: The JastAdd Extensible Java Compiler, *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, New York, NY, USA, ACM Press, 2007, pp. 1–18.

[13] Gonzlez, S., Mens, K., and Cdiz, A.: Context-Oriented Programming with the Ambient Object System, *Journal of Universal Computer Science*, Vol. 14, No. 20(2008), pp. 3307–3332.

[14] Gosling, J., Joy, B., Steele, G., and Bracha, G.: *Java(TM) Language Specification, The 3rd Edition*, Addison-Wesley Professional, 2005.

[15] Haupt, M. and Mezini, M.: Micro-Measurements for Dynamic Aspect-Oriented Systems, *Proc. Net.ObjectDays 2004*, Weske, M. and Liggesmeyer, P.(eds.), Lecture Notes in Computer Science, Vol. 3263, Berlin, Heidelberg, Germany, Springer-Verlag, 2004.

[16] Hedin, G. and Magnusson, E.: JastAdd: An Aspect-oriented Compiler Construction System, *Science of Computer Programming*, Vol. 47, No. 1(2003), pp. 37–58.

[17] Hirschfeld, R., Costanza, P., and Haupt, M.: An Introduction to Context-Oriented Programming with ContextS, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7. 2007, Revised Papers*, Lämmel, R., Visser, J., and Saraiva, J.(eds.), Lecture Notes in Computer Science, Vol. 5235, Berlin, Heidelberg, Germany, Springer-Verlag, 2008, pp. 396–407.

[18] Hirschfeld, R., Costanza, P., and Nierstrasz, O.: Context-oriented Programming, *Journal of Object Technology*, Vol. 7, No. 3(2008), pp. 125–151.

[19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G.: An Overview of AspectJ, *15th European Conference on Object-Oriented Programming, ECOOP 2001*, Knudsen, J. L.(ed.), Lecture Notes in Computer Science, Vol. 2072, Berlin, Heidelberg, Germany, Spinger-Verlag, January 2001, pp. 327–354.

[20] Kiczales, G., Lamping, J., Mendhekar, A.,

Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: Aspect-oriented Programming, *Proceedings 11th European Conference on Object-Oriented Programming*, Vol. 1241, Springer-Verlag, 1997, pp. 220–242.

[21] Klein, G., Rowe, S., and Décamps, R.: JFlex - The Fast Scanner Generator for Java, October v1.4.3 released 01/2009. http://www.jflex.de.

[22] Kniesel, G., Rho, T., and Hanenberg, S.: Evolvable Pattern Implementations need Generic Aspects, research report C-196, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan, June 2004.

[23] Lieberherr, K. J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Co., Boston, MA, USA, 1995.

[24] Madsen, O. L., Mø-Pedersen, B., and Nygaard, K.: *Object-oriented Programming in the BETA Programming Language*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[25] Nokia Corporation: Whitepaper: A Technical Introduction to Qt, 2008. http://www.qtsoftware.com.

[26] Ossher, H. and Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach, *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Akşit, M.(ed.), Kluwer, 2000, pp. 293–323.

[27] Schippers, H., Haupt, M., Hirschfeld, R., and Janssens, D.: An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns, *Proc. SAC PSC*, ACM Press, to appear, 2009.

[28] Schippers, H., Janssens, D., Haupt, M., and Hirschfeld, R.: Delegation-based Semantics for Modularizing Crosscutting Concerns, *ACM SIGPLAN Notices*, Vol. 43, No. 10(2008), pp. 525–542.

[29] Schmidt, G.: ContextR & ContextWiki, Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[30] Schubert, C.: ContextPy & PyDCL - Dynamic Contract Layers for Python, Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[31] von Löwis, M., Denker, M., and Nierstrasz, O.: Context-oriented Programming: Beyond Layers, *ICDL '07: Proceedings of the 2007 International Conference on Dynamic Languages*, Demeyer, S. and Perrot, J.-F.(eds.), ACM International Conference Proceeding Series, Vol. 286, New York, NY, USA, ACM Press, 2007, pp. 143–156.