ContextJ: Context-oriented Programming with Java

Malte Appeltauer　Robert Hirschfeld　Michael Haupt
Hidehiko Masuhara

The development of context-aware systems requires dynamic adaptation that challenges state-of-the-art programming language support. Context-oriented programming (COP) provides dedicated abstractions for first-class representation of context-dependent behavior. So far, COP has been implemented for dynamically-typed languages such as Lisp, Smalltalk, Python, Ruby, and JavaScript relying on reflection mechanisms, and for the statically-typed programming language Java based on libraries and pre-processors. ContextJ is our compiler-based COP implementation for Java that properly integrates COP's layer concept into the Java type system. In this paper, we introduce ContextJ's language constructs, semantics, and implementation. We present a case-study of a ContextJ-based desktop application.

1 Introduction

For the evolution and maintainability of large software applications, the modularization abstractions provided by the programming language of use are crucial factors of success. The benefits of object-oriented modularization aside, some requirements are not met by this paradigm.

For instance, program behavior can depend on execution context information, such as control flow, user information, network accessibility, and more. Based on such context information it can be necessary to dynamically adapt a system. With object-orientation as a foundation, several approaches, e. g., *aspect-oriented programming* [22] and *feature-oriented programming* [8], have been emerged to cope with variability and dynamic adaptation.

Context-oriented programming [19] (COP) is a novel approach to dynamic composition, 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 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 other literature [19]. The approach has been implemented mainly for dynamic languages, such as Lisp [10], Smalltalk [18], Python [31], and Ruby [30].

Except for two mere proof-of-concept implementations of COP for Java [19][5], the approach has not been fully integrated into a statically typed language yet. However, such languages gain increasing relevance for development of Web and desktop applications relying on context information.

Based on our experiences with previous COP implementations, we postulate the following two requirements for a Java language extension. First, our COP extension should be fully integrated into the Java language. This includes an intuitive syntax extension organically merging the COP con-

ContextJ: Java 上の文脈指向プログラミング.
Malte Appeltauer, Robert Hirschfeld and Michael Haupt, Hasso Plattner Institute, University of Potsdam.
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.
コンピュータソフトウェア, Vol.28, No.1 (2011), pp.272–292.
[研究論文] 2009 年 8 月 18 日受付.
大会同時投稿論文

cepts with the Java type system. Second, run-time performance must be considered. Existing implementations of COP extensions to dynamic languages extensively use the languages' core features and meta-level capabilities. Such a library-based prototype for Java [19] is presented in Section 6, however it suffers critical run-time costs (see Section 5. 3). To overcome this issue, we need to use an alternative implementation strategy.

The contributions of this paper are as follows.

- A ContextJ language specification extending Java 1.6 and its comparison to previous approaches for Java.
- A compiler-based implementation that realizes ContextJ's enhanced method dispatch.
- Measurements with micro-benchmarks measuring the performance impact of our language extension.

Our paper is structured as follows. Section 2 motivates first-class representation of behavioral variations. Section 3 gives a brief overview of COP, introduces ContextJ, and explains its features for the modularization and run-time composition of context-dependent concerns. Section 4 presents a case study and compares a ContextJ-based implementation with its AspectJ-based version. The ContextJ compiler is presented in Section 5. Related work is discussed in Section 6. The paper is summarized in Section 7.

2 Behavioral Variations

Context-dependent applications vary their behavior according to conditions arising at run-time. The implementation of such variations can range from `if` conditions in simple cases to dynamic class reloading or recomposition in component-based architectures. Context-specific adaptations can require changes in multiple locations in a system, leading to scattered implementations and code tangling. In any case, an appropriate representation of variations is crucial for software understanding and evolution. In this paper, we focus on behavioral variations and their representation at source code level.

In the following, we discuss different representations of behavioral variations and refer to a simple Java-based bank account example presented in Fig. 1. 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. For quality management purposes, an extensive logging mechanism should be established for transactions. Since logging consumes considerable execution time, it is only applied to samples that are determined using `loggingSample`. Some banks have special security policy agreements for inter-bank transactions. Depending on that policy, which is computed by `securityLevelHigh`, transactions are es-

```
1  public class Account {
2    private int accountNumber;
3    private float balance;
4
5    public Account(int accNr) {
6      accountNumber = accNr;
7    }
8    public void credit(float amount) {
9      balance = balance + amount;
10   }
11   public void debit(float amount) {
12     balance = balance - amount;
13   }
14   public float getBalance() {
15     return balance;
16   }
17 }
18
19 public class TransferSystem {
20   public void transfer(Account from, Account to,
21                        float amount) {
22     from.debit(amount);
23     to.credit(amount);
24   }
25 }
26
27 public class Main() {
28   public void transfer100(Account from,
29                           Account to) {
30      new TransferSystem.transfer(from, to, 100);
31   }
32   boolean securityLevelHigh(Account from,
33                             Account to) { ... }
34   boolean loggingSample(Account from,
35                         Account to) { ... }
36 }
37 public class Encryption { ... }
38 public class Logger { ... }
```

Fig. 1 Bank accounts and transfers.

pecially encrypted. Both concerns, logging and encryption, require behavioral variations of dynamic adaptations of control flow. We will present object-oriented and aspect-oriented implementations of this requirement and discuss their benefits and weaknesses.

2.1 Object-oriented Implementation

A naïve Java-based implementation is shown in Fig. 2. The actual composition of context information is represented by flags in a context object that is passed as first argument to each method. The code of behavioral variations is implemented by `if` statements checking the respective field of the context object; variations are represented by its corresponding block.

The benefit of this solution is its simplicity: composition information can be passed as an argument; each method can access these information an adapt its behavior accordingly. However, some drawbacks need to be discussed. Typically, behavioral variations include adaptations in several objects rather than a single place in a control-flow. Even in our simple example, the encryption concern requires adaptations in four methods. Although these adaptations are semantically related, this relationship cannot be made explicit in object-oriented languages. Developers can only infer from code structure or comments that the first `if` branches in `credit` and `debit` are related. Thus, from a modularization perspective, the proposed implementation suffers from scattered implementations of variations and tangling of core and context-dependent concerns in method bodies.

In addition, the use of context objects requires modifications of any method signature and context argument passing for any method call—tedious tasks that could be implicitly conducted by a more elaborate compiler or execution environment. The aforementioned implementation strategy is only applicable if source code is accessible so that methods can be extended. However, libraries and frameworks only provide bytecode. Thus, adaptations of classes require other, more complex approaches.

```
1  public class Ctx {
2    public boolean encryption, logging;
3  }
4
5  public class Account {
6    ...
7    public void credit(Ctx ctx, int am) {
8      if(ctx.encryption)
9        am = Encryption.decrypt(am);
10     ...
11     if(ctx.logging)
12       Logger.logCredit(this, am);
13   }
14   public void debit(Ctx ctx, int am) {
15     if(ctx.encyption)
16       am = Encryption.decrypt(am);
17     ...
18     if(ctx.logging)
19       Logger.logDebit(this, am);
20   }
21   public int getBalance(Ctx ctx) {
22     int val = balance;
23     if(ctx.encyption)
24       val = Encryption.encrypt(val);
25     if(ctx.logging)
26       Logger.logBalanceRequest(this, val);
27     return val;
28   }
29 }
30
31 public class TransferSystem {
32   public void transfer(Ctx ctx, Account from,
33                        Account to, int amount) {
34     ...
35 }
36 public class Main() {
37   public void transfer100(Account{a,
38                           Account b) {
39     Ctx ctx = new Ctx();
40     ctx.encryption = securityLevelHigh(a,b);
41     ctx.logging = loggingSample(a,b);
42     new TransferSystem.transfer(ctx, a, b, 100);
43   }
44 }
```

Fig. 2 Java-based implementation of behavioral variations.

2.2 Aspect-oriented Implementation

In the Java implementation, semantically related behavioral variations are scattered over the application's decomposition, and tangled with its core concerns. This issue is known as *crosscutting concerns* (CCCs), program behavior that cannot be adequately modularized with respect to the other parts of a system [27]. Such concerns typically hinder software evolution and maintenance. *Aspect-*

oriented programming (AOP) [22] supports modularization of CCCs with dedicated language constructs. In AOP, a CCC consists of 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.

Figure 3 shows an aspect-oriented implementation of the variations in our account example using *AspectJ* [21], an aspect-oriented language extension to Java. Its join point model includes method calls and executions as well as field accesses. Advice blocks introduce additional behavior before, after, or around the join point.

In our implementation, aspects encapsulate context-specific concerns. Pointcuts describe the join points on which variations, represented by advice, should be executed. The dynamic evaluation of the context composition is realized by `if`-pointcuts (Lines 10, 19) that access a thread-local context instance. This context instance must be specified at the beginning of a composition (Lines 27–31).

Although this implementation eases the representation of context-dependent behavior, there are some conceptual issues left. First, the definition of behavioral variations in advice is redundant. Their corresponding pointcuts all have the same structure, but cannot be generalized because of their concrete method signature bindings. Second, composition scope cannot be declared explicitly. Instead, composition start and end are defined programmatically; there is no block construct enforcing scope. This can lead to fragile and inconsistent adaptations if composition statements are not properly declared or executed. Third, variations are defined within aspects, isolating them from their corresponding classes. We argue that behavioral variations should be defined within the scope of their respective object rather than in an external module, and that compositions should be clearly scoped to regions of code and dynamic extent.

To characterize the requirements for appropriate representation of behavioral variations, we distinguish between *homogeneous* and *heterogeneous* CCCs [1]. A homogeneous CCC executes the same

```
1  public class Ctx {
2    public boolean encryption, logging;
3    public static ThreadLocal<Ctx> comp = ...
4  }
5
6  public aspect Encryption {
7    void around(int am) :
8      execution(void Account.credit(int)) &&
9      args(am) &&
10     if(Ctx.comp.get().encryption) {
11     proceed(Encryption.decrypt(am));
12   }
13   ...
14 }
15 public aspect Logging {
16   void around(int am) :
17     execution(void Account.credit(int)) &&
18     args(am) &&
19     if(Ctx.comp.get().logging) {
20     Logger.logCredit(this, am);
21   }
22   ...
23 }
24
25 public class Main() {
26   void transfer100(Account a, Account b) {
27      Ctx ctx = new Ctx();
28      ctx.encryption = securityLevelHigh(a,b);
29      ctx.logging = loggingSample(a,b);
30      Ctx previous = Ctx.composition.get();
31      Ctx.composition.set(ctx);
32      new TransferSystem.transfer(a, b, 100);
33      Ctx.composition.set(previous);
34   }
35 }
```

Fig. 3 AspectJ-based implementation of behavioral variations.

functionality in multiple locations in a program's control flow graph. A typical example of such an concern is simple logging, where the same functionality (e. g., writing system statistics into a file) occurs on several places in a system (e. g., every time a database is queried or a user executes an action). AOP provides well suited abstractions to model such CCCs; however, behavioral variations may introduce completely different behavior at multiple locations of a program, which we denote as heterogeneous CCCs. Even in our simple example, the logging concern requires different functionality at each join point it applies to. Thus, the biggest

benefit of AOP, namely the declarative description of one-to-many relations of source code, does not affect our application.

If composition is applied statically at compile time, concepts of *feature-oriented programming* [8] (FOP), can be employed. FOP introduces the concept of *layers* and their composition, which will be explained in the following section. However, FOP focuses on static composition, thus it is inappropriate for the representation of context-dependent behavioral variations.

3 Context-oriented Programming for Java

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 of 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.

Figure 4 (left) illustrates modularization with layers. Each layer provides its behavioral variations while preserving the object-oriented decomposition. Contrary, the AOP approach fully encapsulates CCCs and declaratively specifies variation points within an application, as shown in Figure 4 (right). The main distinction between AOP and COP is that the former allows for a joint specification of *when* in the execution flow *what* kind of functionality should be used, while COP separates *when* (using explicit composition scopes) from *what* (using layers and partial methods). Most AOP languages can mimic features of COP using pointcuts and advice, though in an unwieldy manner.

3.1 ContextJ

We discuss ContextJ's language features along the implementation of encryption and logging in our example. 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 [15].

3.1.1 Modularization

Layer. ContextJ extends the Java type system with layers, special non-instantiable types, and provides the *layer-in-class* style [19]; 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.

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

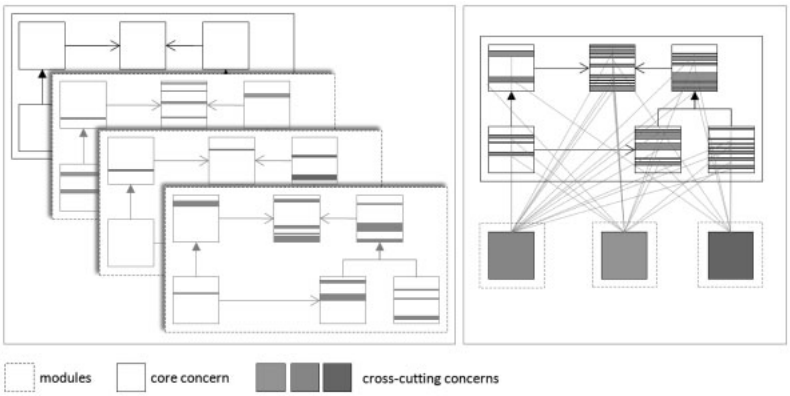

Fig. 4 Modularization techniques. ***left:*** **COP,** ***right:*** **AOP.**

method of the enclosing class or its superclass. Final methods cannot be extended by layers.

Layer Type. Layers are referenced by layer type 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* `)`

Figure 5 depicts the separate declaration of two layers that implement crosscutting concerns. For example, the definition of `EncryptionLayer` in `Account` (Lines 6–16) 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 34–40).

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) 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).

3.1.2 Dynamic Composition

Layer Activation. To control scoped layer activation, ContextJ introduces a new block statement,

```
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(Encryption.decrypt(am));
9      }
10     public void debit(int am) {
11       proceed(Encryption.decrypt(am));
12     }
13     public int getBalance() {
14       return Encryption.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 public class TransferSystem {
33   ...
34   layer LoggingLayer {
35     after public void transfer(Account from,
36                                Account to,
37                                int amount) {
38       ...
39     }
40   }
41 }
```

Fig. 5 Layers for encryption and logging.

`with`, that can be used in method bodies. The `with` block provides an argument list that contains layer type expressions denoting the layers to be activated. More precisely, expressions of type `Layer`, `Iterable<Layer>`, or `Layer[]` are valid arguments; the usage of any other type will cause an runtime exception. If all `with` arguments are evaluated to an empty list (or `null`), no layer will be activated.

The specified layers are only active for the *dynamic extent* of the `with` block. This implies that the activation of a particular layer is confined to the threads in which the layer was explicitly activated. Layer activation does not propagate to new threads; they start with no layers being active.

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.

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.

Figure 6 presents different layer compositions in our account example. The nested composition activates `LoggingLayer` and a list of layers returned by `transferComposition` (Lines 4–7). Lines 8–10 contain another activation using a list of layer identifiers in a single `with` block.

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**}

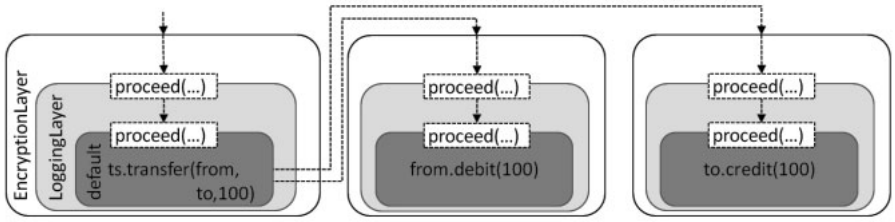

Fig. 7 Progression of a method invocation through a list of active layers.

```
1  public class Main() {
2     ...
3    void transfer100(Account from, Account to) {
4      with(LoggingLayer) {
5        with(transferComp(from, to)) {
6          transferSystem.transfer(from, to, 50);
7      }
8      with(LoggingLayer, EncryptionLayer) {
9        transferSystem.transfer(from, to, 50);
10     }
11   }
12
13   List<Layer> transferComp(Account from,
14                            Account to) {
15     List<Layer> layers = new ArrayList<Layer>();
16
17     if(securityLevelHigh(acct))
18       layers.add(EncryptionLayer);
19     if(loggingSample(a))
20       layers.add(LoggingLayer);
21     return layers;
22   }
23 }
24
25 public class TransferSystem {
26   layer EncryptionLayer {
27     public void transfer(Account from,
28                          Account to, int am) {
29       without(LoggingLayer) {
30         proceed(from, to,
31           Encryption.encrypt(am));
32       }
33     }
34   }
35 }
```

Fig. 6 Different kinds of layer activation.

Figure 6 (Lines 27–33) contains a partial method declaration of `transfer` that uses `without` to prevent the logging layer from monitoring the transaction.

Layer Composition. Figure 7 illustrates the execution of `transfer` in Lines 6 and 9 in Listing 6. 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 7.

3.1.3 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`. Fig. 8 presents an implementa-

Table 1 The ContextJ reflection API.

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

```
1  layer EncryptionLayer {
2    public void transfer( ... ) {
3      Composition comp = Composition.current();
4      Layer[] ls = comp.getLayers();
5      String sig = /* this method's signature */;
6      for(Layer l : ls ) {
7        if((l != EncrpytionLayer) &&
8           (l.providesPartialMethodFor(sig)))
9          throw new RuntimeException(
10           "A layer must not override a method.");
11     }
12     /* do the encryption */
13   }
```

Fig. 8 Use of reflection API.

tion of such behavior. First, we access the current composition (Line 3) and retrieve an array 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 Case Study

Interactive development environments (IDEs) nowadays provide a large feature set for editing and managing source code, including specific editors for file-based, source code-based, or debugging based representation of a program, imposing considerable complexity on developers. To ease the use of complex work-flows, IDEs often offer context-specific perspectives that emphasize important and hide irrelevant functionality.

We have developed *CJEdit*, a little IDE whose GUI provides context-specific user interface (UI) behavior. *CJEdit* is a simple programming environment for ContextJ that provides behavioral variations for the tasks *programming* and *documenting*. It supports syntax highlighting, an outline view, a compilation/execution toolbar and rich text commenting features, such as font and color modifications. In addition, the IDE relieves the user from manually switching perspectives and automatically changes them depending on the actual context of use. The UI is recomposed upon these context switches, which are triggered whenever the text cursor moves from text to code blocks and vice versa. The creation of new text and code blocks can be declared by the developer using a toolbar

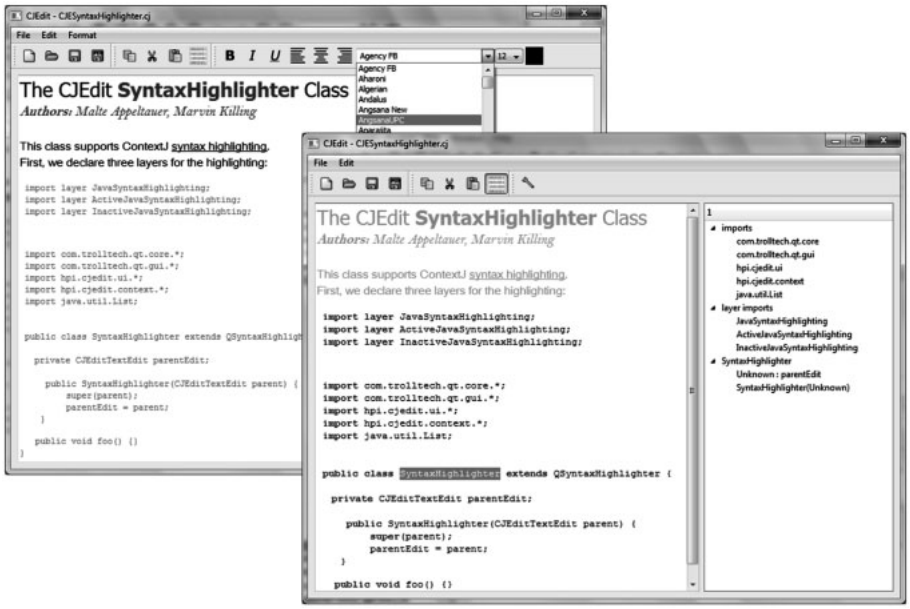

Fig. 9 Context-dependent GUI compositions in CJEdit.

button. Figure 9 shows two screenshots of CJEdit's GUI compositions.

The editor's underlying document tree represents each text line as a text block node. Each block provides information about its type (code node or comment node). The application is recomposed and redrawn whenever the type of the focused block changes from rich text to code block, and vice versa. This change is explicitly activated by entering or leaving the programming activity (by pressing the code button) or on moving the text cursor between blocks of different types. The composition is triggered by the `onCursorPositionChanged` event. The method `getCurrentBlockActivity` returns a String representation of the focused node type and is used to determine node type changes.

CJEdit's core is implemented using Java and the *Qt Jambi GUI Framework* [26]. The editor consists of approximately 3,500 lines of code in 38 classes. In previous work [4], we presented a ContextJ based implementation of CJEdit's context-aware functionality. Here, we compare a more elaborate ContextJ implementation with an AspectJ version, since both approaches provide multi-dimensional separation of concerns, as motivated in Section 2. We will focus on the implementation of the `CJEWindow` class, which is responsible for handling GUI composition. For brevity we present only two layers with few partial methods.

4.1 AspectJ Implementation

In our AspectJ implementation (see Figure 10), we separate the definition of behavioral variations from the base program. Partial method definitions are represented by advice to which execution pointcuts are bound that specify the method's signatures (Lines 25–34). Auxiliary members can be introduced via inter-type declarations (Lines 20–23). Dynamic activation is controlled by `if` pointcuts that must be declared for any advice (Lines 27, 32). It restricts advice execution to join points within the control flow of `onCursorPositionChanged` at which the focused text block type describes the aspect's concern. For dynamic activation, a thread-local composition list has to be maintained. Most of the adapted methods are private. Thus, our aspect requires privileged access to private members and therefore breaks encapsulation.

The complete implementation for CJEdit consists of five concrete aspects (layers) and one abstract aspect (providing auxiliary methods). They provide advice (behavioral variations) for nine methods in three classes. In addition, a context class represents the actual composition and man-

```
1  public class CJEWindow extends QMainWindow {
2    ...
3    // definition of core concerns
4    private void drawToolBars() { ... }
5    private void drawMenus() { ... }
6
7    // dynamic composition
8    void onCursorPositionChanged() {
9      if (blockTypeChanged()) {
10       Ctx old = Ctx.composition.get();
11       Ctx.composition
           .set(getCtxOfCurrentBlock());
12       drawWidgets();
13       Ctx.composition.set(old);
14     }
15   }
16 }
17
18 pivileged public aspect ProgrammingActivity {
19   // auxiliary member inter type declarations
20   private CodeToolBar CJEWindow.codeToolBar;
21   private Menu CJEWindow.codeMenu;
22   private CodeToolBar CJEWindow
                        .createToolBar() {...}
23   private Menu CJEWindow.createMenu() { ... }
24   // definition of behavioral variations
25   after(CJEWindow obj):
26     execution(* *.drawMenus()) &&
27     if(Ctx.comp.get().programming)) {
28       ...
29   }
30   after(CJEWindow obj):
31     execution(* *.drawToolbars()) &&
32     if(Ctx.comp.get().programming)) {
33       ...
34   }
35 }
36
37 privileged public aspect CommentingActivity {
38   ...
39 }
40
41
42 public class Ctx {
43   public boolean programming, commenting;
44   public static ThreadLocal<Ctx> comp = ...;
45 }
```

Fig. 10 Dynamic composition with AspectJ.

ages a thread local composition list. The implementation consists of about 740 lines of code.

```
1  public class CJEWindow extends QMainWindow {
2    ...
3    // definition of core concerns
4    private void drawToolBars() { ... }
5    private void drawMenus() { ... }
6
7    // dynamic composition
8    void onCursorPositionChanged() {
9      if (blockTypeChanged()) {
10       with (getLayersOfCurrentBlock()) {
11         drawWidgets();
12       }
13     }
14   }
15   layer ProgrammingActivity {
16     // definition of behavioral variations
17     after private void drawToolBars() { ... }
18     after private void drawMenus() { ...}
19     // auxiliary members
20     private CodeToolBar codeToolBar;
21     private Menu codeMenu;
22     private CodeToolBar createToolBar() { ... }
23     private Menu createMenu() { ... }
24   }
25   layer CommentingActivity {
26     ...
27   }
28 }
```

Fig. 11 Dynamic composition with ContextJ.

```
1  aspect Authentication {
2    pointcut authMethod() :
3      execution(void CJEditWindow.onPrint()) ||
4      execution(void CJEditWindow.onFileOpen()) ||
5      execution(void CJEditWindow.onFileSave());
6
7    around() : authMethod() {
8      if (isLoggedIn(CJEdit.getUser()))
9        proceed();
10     else
11       throw new AuthorizationException();
12   }
13 }
```

Fig. 12 Dynamic composition with ContextJ.

4.2 ContextJ Implementation

Figure 11 shows the implementation of the *programming* activity-specific widgets using ContextJ layers (Lines 15–27). By default, text blocks refer

to the layers responsible for *rich text commenting* behavior. If the user switches to the *programming* activity (by pressing the code button in the toolbar), subsequently created text blocks are linked with programming environment-specific layers.

The application is recomposed and its GUI redrawn whenever the current block type switches. The dynamic composition of our previously specified layers is depicted in Lines 10–12. For layer composition, ContextJ provides a `with` statement that specifies the layers to be activated, and the dynamic extent for which the composition is valid. Recomposition can be triggered by the `onCursorPositionChanged` event handler that checks if the block type of the previously focused block is different to that of the current block. If so, the method calls `drawWidgets` to update the UI using the current block's layer composition.

The ContextJ implementation consists of five layers spanning over three classes that provide behavioral variations for nine methods. Layer representation requires 400 lines of code.

4.3 Discussion

The ContextJ implementation is more concise than the AspectJ solution. In our examples, the total number of lines of code is significantly reduced, which is due to ContextJ's provision of appropriate abstractions for composition variations. Since the adapted methods are private members of `CJEWindow`, it is also more natural to define their variations within the same lexical scope as with ContextJ, instead of in an external aspect.

For heterogeneous concerns, ContextJ is better suited than AspectJ, as our case study shows. In turn, for the encapsulation of homogeneous crosscuts—if they constitute a one-to-many relationship between adaptation code and its locations in an execution flow—AspectJ has some benefits.

Security concerns in CJEdit are a good example of homogeneous crosscutting. The application offers user management functionality that controls printer and file access. To ensure that only logged-in users are able to open, save, and print files, identical security logic must be implemented at different source code locations. Figure 12 sketches an AspectJ aspect providing this behavior. A pointcut describes the methods requiring authentication (Lines 2–5). Authentication logic itself is encapsulated in an advice (Lines 7–12). Since ContextJ does not support quantification, three redundant partial methods are needed (one per method requiring authentication) that implement the aforementioned around advice. Encapsulation of quantification is subject of ongoing work on ContextJ.

5 Implementation

We developed a compiler for ContextJ because the reflection-based implementation approaches (see Section 6) taken for COP extensions to dynamic languages are not suitable for Java.

5.1 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 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

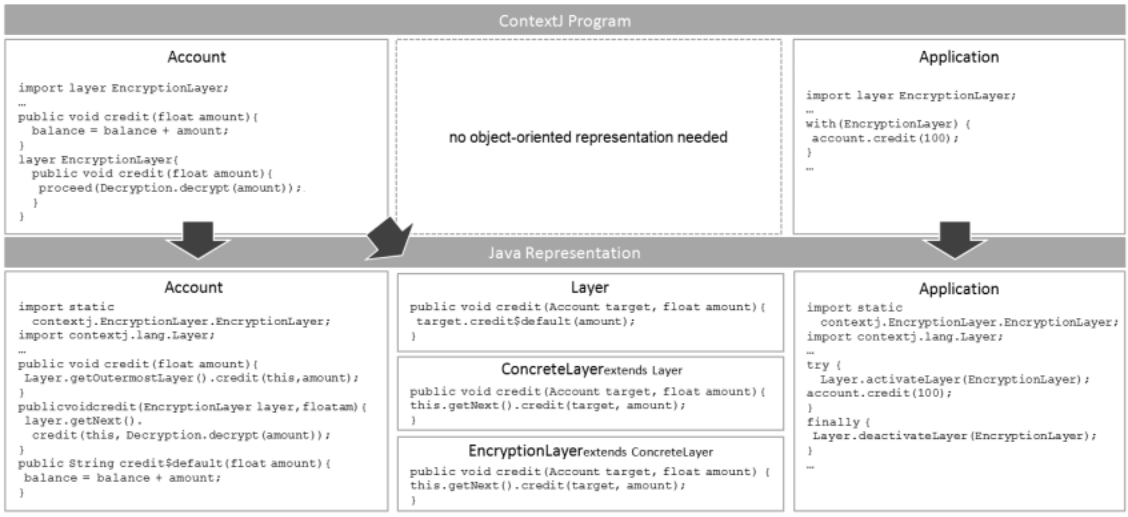

Fig. 13 Mapping of a ContextJ program to Java.

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 13. Note that the Java source code presented here is not generated but directly transformed into byte code during compilation.

5.2 Compiler Implementation

The implementation of the ContextJ compiler is an extension of *JastAddJ* [13], an open Java compiler based on the *JastAdd* [17] 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 dedicated modules. During the compiler build process, the separate specifications are woven into one executable compiler.

For lexical analysis, JastAdd employs *JFlex* [23], 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* [12], 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.

For the implementation of the behavior shown in Section 5.1, we make use of JastAdd's re-writing facilities. Typically, re-write rules change a certain AST node or subtree, or replace it 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.

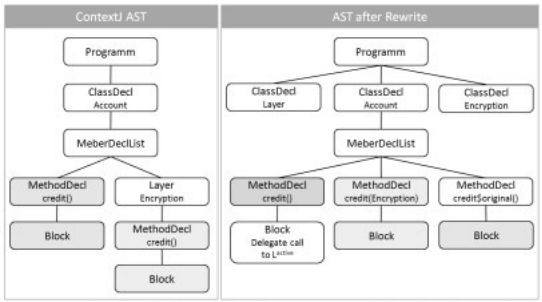

Fig. 14 AST transformation of ContextJ nodes to Java nodes.

1. For each layer L, a class L^{class} will be created as a subtype of `contextj.lang.Layer`
2. 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.
3. 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.
4. The body of a base method M^{base} is moved to a new method M_base.
5. 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.
6. 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 14 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.

5.3 Benchmarks

This section discusses our run-time measurements, based on the *Java Grande Forum Benchmark Suite* [9], for which we developed, in the fashion of [16], 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 for warm-up before the actual measurement to assure that the execution environment is in steady state—i. e., optimizations have been applied—when results are collected.

Below, we will first describe each of the three different measurements we applied. The section is then concluded by a discussion of the various results.

5.3.1 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 *self calls* and *calls to another object* of *synchronized* and *non-synchronized instance* and *class* methods.

The benchmark applies two flavors of plain Java methods. The first, called *implicit composition*, checks a thread-local composition object for context presence and thus requires synchronization and locking. The second, called *parameterized methods*, extends the interfaces of all involved methods by one parameter that carries context information and can be queried for it (see Figure 2. 1).

Figure 15 (top) illustrates the results of this benchmark. If calls are sent to synchronized methods, none of the three different approaches excels, as synchronization is an expensive operation. As they avoid synchronization altogether, plain parameterized methods are significantly faster than their layered counterparts, but also than plain methods using implicit composition. By trend, implicitly composed plain methods perform worse than layered methods, even though both apply synchronization to check thread-local state. The reason for layered methods' better performance is that the ContextJ compiler generates code that uses virtual methods instead of `if/else`-style conditionals, which can be better optimized by the virtual machine.

5.3.2 Layer-aware Message Dispatch

Another set of benchmarks measures the overhead caused by the execution of an increasing num-

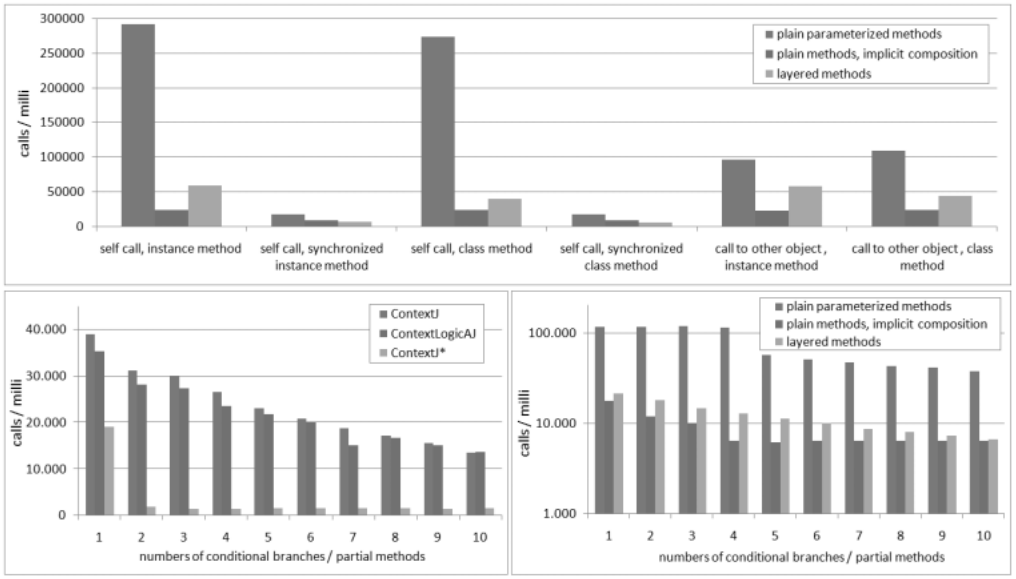

Fig. 15 Micro benchmarks of ContextJ. ***top:*** **Execution of different method types.** ***bottom:*** **Execution time with increasing number of behavioral variations;** ***left:*** **for different Java-based COP implementations;** ***right:*** **for plain Java solutions compared to ContextJ (logarithmic scale).**

ber of partial methods. We again compare the throughputs of plain methods and layered methods. The measurement consists of a plain method and ten integer fields (`c01`–`c10`). The method contains ten nested `if` branches, where each branch increments one field, so that, if a control flow covers all ten nested branches, all fields will have been incremented afterwards. The two strategies for conditional branching mentioned above (implicit composition and parameterized methods) are applied again in this setting. The benchmark version using layers contains one base method `m` that increments `c01`, and nine 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 Fig. 15 (bottom right; note the logarithmic scale). The layered method throughput decreases with an increasing number of layers from approximately 40,000 to 10,000, which is a performance decrease of 75 %. Plain method call throughputs 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 %, with overall performance being about one order of magnitude larger.

5. 3. 3 Java COP Implementations

The preceding measurements compare the runtime behavior of ContextJ with Java. Since one goal of our compiler-based implementation is to provide a competitive 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 Fig. 15 (bottom left).

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.

5.3.4 Discussion

In the following, we will discuss two aspects of the results presented above. On the one hand, the observed ContextJ performance characteristics will be regarded as opposed to plain Java implementations, and on the other, as opposed to other COP implementations' characteristics.

Comparing layered methods to plain Java implementations, the former exhibit significant performance downgrades. 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; e. g., it is conceivable to adopt an approach similar to parameterized methods (cf. Sec. 5. 3. 1).

The performance data we have collected result from running micro-measurements, so they have to be taken with a grain of salt. In a micro-measurement application, the mechanism whose performance is to be assessed occurs in relative isolation: it is not observed in a real application environment. Thus, performance results typically look better or worse than they can be expected to be if the mechanism in question was put to use before the background noise of application logic. Moreover, micro-measurements do not deliver fully accurate results, as they have a strong tendency towards measuring the capabilities of the used virtual machine's just-in-time compiler, instead of the performance of the mechanism under consideration.

As of this writing, there is no full-fledged *benchmark suite* for COP languages available. Such a benchmark suite also goes beyond the scope of this paper; we consider it an important building block of future work.

In the comparison of ContextJ with other COP implementations for Java, it is apparent that ContextJ and ContextLogicAJ are almost on par. We argue that ContextJ still has an advantage over ContextLogicAJ, due to increased declarativeness of context-specific variation descriptions.

6 Related Work

In this section, we discuss existing COP language extensions with emphasis on previous approaches for Java. In addition to the alternative Java-based and AspectJ-based implementations of behavioral variations presented in Sections 2 and 4, we investigate other aspect-oriented languages and feature-oriented systems.

6.1 Context-oriented Programming

COP has been implemented for several host languages and adopted to their host language-specific requirements. We give an overview of these implementations, in particular of Java-based predecessors of ContextJ.

6.1.1 COP for Dynamic Languages

ContextL [10][11] was the first COP extension to a programming language. It is based on Lisp and extends the Common Lisp Object System (CLOS). 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* [18] for Smalltalk, *ContextR* [30] for Ruby, *ContextJS* for JavaScript, *ContextPy* [20] and *PyContext* [31] for Python, and *ContextG* for Groovy. A minimal subset of ContextJ, *cj*, is implemented for the *delMDSOC* kernel [29].

Another approach to context-orientation is *Ambience* and its underlying *Ambient Object System* [14] (AmOS). AmOS is a prototype-based object system built on top of Common Lisp that supports behavioral adaptations with partial method definitions and context objects, which correspond to COP layers. At any method call in AmOS, receiver methods are first looked up in the current activation and then in further enclosing lexical scopes. If no appropriate method is found in the lexical scope, the lookup continues in a graph of context objects delegating to each other. The delegation chain between these context objects can be modified dynamically, achieving context-specific behavior.

These context-oriented extensions are implemented using the respective language's meta-level facilities; none of them utilizes bytecode transfor-

mation as ContextJ does. Few of them (most notably, ContextL and *cj*) provide syntactic means for layers and layer composition; most express both constructs by existing means. A dedicated syntax such as that of ContextJ eases static program analysis and allows for meaningful error messages at compile-time. In addition, static analysis can be employed for compile-time optimizations. A thorough comparison of COP languages and their features is provided in [2].

6.1.2 COP for Java

The first ideas about a ContextJ language have been presented in [11] to improve the accessibility of the *ContextL* code discussed in that paper. The authors introduced ContextJ syntax only in a pseudo-code manner and neither provided a feature-complete syntax nor a language specification, let alone a full implementation. Nevertheless, a proof-of-concept implementation called *ContextJ** [19] exists.

This Java 1.5 library implements the core concepts of COP, i.e., layer definition and activation without any extension to the syntax or semantics of the language. Figure 16 exemplifies layer declaration using ContextJ* by an implementation of our account example.

Concrete layers are represented by subclasses of `Layer` (Lines 2–3). Partial methods are defined in a `LayerDefinitions` container (Lines 22–37). Each layer declaration is a pair of a layer class references and an anonymous class that specifies its partial methods. The base methods (Lines 13–21) execute the layer aware lookup by calling `LayerDefinitions.select`. Layer composition uses the static method `with` followed by a structure of method calls and anonymous class definitions (Lines 42–45). For more details about ContextJ* and its usage, we refer to [19].

As this example illustrates, layers and partial methods can be defined independently of base methods. However, the proper use of ContextJ* requires developers to write boilerplate code adhering to the following idioms:

- Classes providing partial methods must implement a specific interface guaranteeing at least the signatures of layered methods. Whenever a new partial method is defined, this interface must be modified (Lines 6–7, 9).
- Each Layer must provide partial method definitions for all methods of this interface, even if it does not intend to change the method's behavior (Lines 23, 27, 33).

```
1  public class Layers {
2    public static Layer Encryption = new Layer();
3    public static Layer Logging = new Layer();
4  }
5
6  public interface IAccount { ... }
7  public interface ITransfer { ... }
8
9  public class Account implements IAccount {
10   private LayerDefinitions<IAccount>layers =
11     new LayerDefinitions<IAccount>();
12
13   public void credit(int am) {
14     layers.select().credit(am);
15   }
16   public void debit(int am) {
17     layers.select().debit(am);
18   }
19   public int getBalance() {
20     return layers.select().getBalance();
21   }
22   { layers.define(RootLayer,
23       new IAccount() {
24         public void credit(int am) { ... }
25     });
26     layers.define (Layers.Encryption,
27       new IAccount() {
28         public void credit(int am) { ... }
29         public void debit(int am) { ... }
30         public int getBalance() { ... }
31     });
32     layers.define(Layers.Logging,
33       new IAccount() {
34         public void credit(int am) { ... }
35         public void debit(int am) { ... }
36         public int getBalance() { ... }
37     });
38 } }
39
40 public class Main {
41   void transfer100(Account a, Account b) {
42     with(Layers.Encryption).eval(new Block() {
43       public void eval() {
44         new TransferSystem.transfer(a, b, 100);
45     }});
46 } }
```

Fig. 16 Bank account implementation using ContextJ*.

```
1  public class EncryptionLayer extends Layer {}
2  public class LoggingLayer extends Layer {}
3
4  public class Account {
5    ...
6    void credit(EncryptionLayer ctx, int am) {
7      proceed(Encryption.decrypt(am));
8    }
9    void debit(EncryptionLayer ctx,int am) {
10     proceed(Encryption.decrypt(am));
11   }
12   int getBalance(EncryptionLayer ctx) {
13     return Encryption.encrypt(proceed());
14   }
15   void credit(LoggingLayer ctx, int am) {
16     proceed(am);
17     Logger.logCredit(this, am);
18   }
19   void debit(LoggingLayer ctx, int am) {
20     proceed(am);
21     Logger.logDebit(this, am);
22   }
23   int getBalance(LoggingLayer ctx) {
24     int balance = proceed();
25     Logger.logBalanceRequest(this, balance);
26     return balance;
27   }
28 }
29
30 public class Main() {
31   void transfer100(Account a, Account b) {
32     activateLayer(EncryptionLayer.class);
33     new TransferSystem.transfer(a, b, 100);
34     deactivateLayer(EncryptionLayer.class);
35   }
36 }
37 public class Encryption {}
38 public class Logger {}
```

Fig. 17 Bank account implementation using ContextLogicAJ.

- Base methods must manually trigger layer selection (Lines 14, 17, 20).
- Layer activation requires the generation of an anonymous class `Block` whose (Lines 42–46) `eval` method contains the actual code.

All these guidelines required by the library increase code fragility. In the following, we describe a pre-compiler that was developed based on an aspect-oriented language; this compiler overcomes some of these issues.

ContextLogicAJ [5][3] is an aspect-oriented precompiler that offers more convenient layer declaration constructs than ContextJ*. It is based on a *LogicAJ* [24] aspect library. As in ContextJ*, layers are represented by subclasses of a `Layer` class, as shown in Figure 17 (Lines 1–2). Partial method declarations are distinguished by the type of their first parameter, which represents their corresponding layer (Lines 6–27). Calls of the static method `proceed` are join point hooks for ContextLogicAJ's aspect that takes care for the correct method lookup. Layer composition is controlled with (`de`)`activateLayer` (Lines 32, 34). ContextLogicAJ does not provide a scoped activation but expects the developer to explicitly deactivate layers at the end of a composition.

In comparison to ContextJ*, partial methods can be defined more conveniently, but still some idioms, such as declaring a dummy layer class, parameterizing methods with layer types, and explicitly (de)activating, need to be considered. ContextJ abandons these idioms, adopting first-class layers and layer composition.

6.2 Aspect-oriented Programming

Throughout this paper, we discussed the representation of behavioral variations and compared AspectJ- and ContextJ-based implementations. The main concern of AOP languages is the declarative description of control-flow graph locations at which certain pieces of functionality should be executed (see Section 2.2). Aspect *weaving*, as the adaptation process is called in AOP, can be applied at compile-, load-, or run-time. Classic AOP languages such as AspectJ only support static weaving at compile- and load-time, whereas COP languages explicitly target dynamic adaptation.

Some aspect-oriented languages, such as *CaesarJ* and *JAC* also support dynamic weaving. CaesarJ [6] comes with an alternative module concept by unifying classes, aspects, and packages. Its aspects can be deployed at run-time using different kinds of dynamic scope, much like ContextJ layers. The language supports virtual classes [25], 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.

JAC [28] (Java Aspect Components) is an AOP framework supporting dynamic weaving. JAC is based in Javassist, a meta-programming framework for Java. It does not require a language extension. Instead, aspects are represented by objects. Aspect methods can wrap application methods (advice) or introduce new methods (inter-type declarations). Run-time aspect composition is managed by a wrapping controller object.

Dynamic weaving as supported by CaesarJ and JAC allows for controlling and scoping aspect-based adaption at run-time. The aspect-oriented version of our ongoing example presented in Figure 3 can be enhanced by aspect deployment scope, much like ContextJ's `with` statement. However, the aforementioned conceptual differences remain.

AOP aims to tame crosscutting concerns by introducing pointcut-based quantification. Most behavioral variations, however, are heterogeneous crosscuts that require different functionality at each join point; a declarative description of join points is not necessary. In that regard, AOP can be applied for behavioral variations, but introduces unnecessary complexity. From a modularization perspective, a major distinction of the presented Java-based aspect languages and ContextJ is the source location of partial method definitions. ContextJ supports *layer-in-class* declaration and therefore differs from aspect-oriented encapsulation.

6.3 Feature-oriented Programming

Feature-oriented programming (FOP) [8] addresses the process of step-wise refinement for product-line development. The Java-based AHEAD Tool Suite [7] 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.

7 Summary and Conclusion

The modularization of dynamic adaptation is a well known topic that is addressed by several programming paradigms and language extensions. To assess their usability and expressiveness, these approaches need to be applied to different language domains. 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 ContextJ, a context-oriented programming language extension to Java. ContextJ provides first-class support for layers and constructs for their dynamic composition. Layers are integrated into the Java type system as non-instantiable types and can be referred to like common Java types. We describe modularization and dynamic composition of layers and their behavioral variations. We show the design and implementation of our ContextJ compiler and its layer-aware method lookup. In a case study, ContextJ is applied to the implementation of a context-aware programming environment containing several heterogeneous crosscutting concerns. In comparison to an alternative AspectJ-based implementation, we identify some advantages of our layer-based approach for representing these specific crosscuts.

In future work, we will continue to apply ContextJ to several problem domains for dynamic context-specific adaptations and analyze the expressiveness of the abstractions ContextJ provides.

Acknowledgments

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

References

[1] Apel, S., Leich, T. and Saake, G.: Aspectual Feature Modules, *IEEE Transactions on Software Engineering*, Vol. 34, No. 2(2008), pp. 162–180.

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

[3] Appeltauer, M. and Hirschfeld, R.: Explicit Language and Infrastructure Support for Context-aware Services, in *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.

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

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

[6] 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.

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

[8] 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.

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

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

[11] Costanza, P., Hirschfeld, R. and De Meuter, W.: Efficient Layer Activation for Switching Context-dependent Behavior, in *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.

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

[13] Ekman, T. and Hedin, G.: The JastAdd Extensible Java Compiler, in *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.

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

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

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

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

[18] Hirschfeld, R., Costanza, P. and Haupt, M.: An Introduction to Context-Oriented Programming with ContextS, in *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.

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

[20] Hirschfeld, R., Perscheid, M., Schubert, C. and Appeltauer, M.: Dynamic Contract Layers, in *25th Symposium on Applied Computing, Lausanne, Switzerland*, New York, NY, USA, ACM DL, 2010.

[21] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, in *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.

[22] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented Programming, in *Proceedings 11th European Conference on Object-Oriented Programming*, Vol. 1241, Springer-Verlag, 1997, pp. 220–242.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java, in *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, London, UK, Springer-Verlag, 2001, pp. 1–24.

[29] Schippers, H., Haupt, M., Hirschfeld, R. and Janssens, D.: An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns, in *Proc. SAC PSC*, ACM Press, 2009, pp. 1944–1951.

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

[31] von Löwis, M., Denker, M. and Nierstrasz, O.: Context-oriented Programming: Beyond Layers, in *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.